A Dynamic Majority Determination Algorithm for Reconfiguration of Network Partitions

BHARAT BHARGAVA

and

PETER LEI NG

Department of Computer Sciences, Purdue University, West Lafayette, Indiana 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 nonblocking 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 avoids any such merge condition and integrates the partitions whenever a 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 stored at a set of sites interconnected by a communication network. The granularity of these logical data can be a record, a relation a file, etc. Without loss of generality, in the following discussion, we assume the granularity of these data 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
A DYNAMIC MAJORITY DETERMINATION ALGORITHM

1.2. DISCUSSION OF THE RESEARCH PROBLEM

The research problem is to find solutions to allow:

1. read access to the latest copy on all sites,
2. the determination of 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 designating majority of the previous majority as the new majority. Of course, any site can join the majority partition. The update access is restricted so 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 acceptable 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.

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 consistence of a file is a sufficient condition for the correct access of that file, maintaining such mutual consistence 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 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, for 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. A 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 reconcile 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. 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). No other copies of that file in other partitions are 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 rollbacks of transactions are needed. The research in this paper contributes to the conservative approach.
EXAMPLE 1. Consider the partition history of a file represented by the partition graph [15] in Figure 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.

In 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. But we do have a choice here. This issue of choice is discussed further in 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.

A DYNAMIC MAJORITY DETERMINATION ALGORITHM

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 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), then 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 low, 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 an 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 the 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 the update sites vector (SV) is used instead of the integer version as in [11]. Even though simpler data structures reduce the maintenance effort, in 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 tradeoff—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 merging of sites. Even if sites are physically connected, they pretend that such sites are in different partitions. 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

2.1. 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, \( s_1, s_2, s_3 \), then 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 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.

2.2. 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 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 \( S = \rho(s_1, s_2, \ldots, s_n) \), is a vector of the names (or identification numbers) of the \( n \) sites at which \( f \) is replicated. The linear order vector of \( f \), denoted by \( L = \langle l_1, l_2, \ldots, l_n \rangle \), 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 pair of values of \( S \) and \( L \) could be \( S = \langle A, B, C \rangle \) and \( L = \langle 1, 2, 3 \rangle \), 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 they 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 \( V = \langle v_1, v_2, \ldots, v_n \rangle \), 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 Example 1, let the version vector at site \( s_i \) have a value of \( V = \langle 2, 0, 0 \rangle \). 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 \( V = \langle 2, 0, 2 \rangle \).

The marker vector of a copy, denoted by \( M = \langle m_1, m_2, \ldots, m_n \rangle \), 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).

2.3. 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 the rule 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 the 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 occurrence of this partition. 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 mechanism).

**Procedure PARTITION** (assume invoked at site $A$)

Input: $C$ - The connection vector,

$X_1, X_2, \ldots, X_k$ - version numbers of the $k$ copies of replicated sites $f_1, f_2, \ldots, f_k$,

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

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

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

Method: For each $V_i, 1 \leq i \leq k$, for each $V_j$ in $V_i$, do the following: if $S_i[j]$ is still connected with $A$, or is separated from $A$ before this partitioning (value $V_i[j] = 0$), 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 $X_j$.

**Procedure PARTITION**($C, X_1, X_2, \ldots, X_k, S_1, S_2, \ldots, S_k, V_1, V_2, \ldots, V_k$):

```
begin
  let $X$ = $\max\{X_i|V_i = V\}$;
  if $X < X_i$ then return "yes"
  else $X = X_i$ /$^+$

  begin
    $\text{Set1} \leftarrow \{S_i|V_i = 0\}$ and $M_i$ is $\bot$
    $\text{Set2} \leftarrow \{S_i|V_i = 0$ and $M_i$ is $\top\}$;
    if $|\text{Set1}| > |\text{Set2}|$ then return "yes"
    else $|\text{Set1}| = |\text{Set2}|$ and there exists a site $S_i$ in $\text{Set1}$ such that $S_i[j] > S_i[j]$ for all $S_i[j]$ in $\text{Set2}$ /$^*$ by consulting $L$ /$^*$
    then return "yes"
    else return "no"
  end
end.
```

![Fig. 2. Definition of procedure ISMAJORITY.](image)

A Dynamic Majority Determination Algorithm

![Fig. 3. Definition of procedure PARTITION.](image)
Procedure \textsc{resolve} (invoked at the site that initiates the merge)

\textbf{Input:} \(X \subseteq X_1, X_2, \ldots, X_k\) - version members to resolve.
\(V_1, V_2, \ldots, V_k\) - version vectors in resolve.
\(M_1, M_2, \ldots, M_k\) - master vectors to resolve.

\textbf{Output:} \(X\) - the new version number.
\(V\) - the new version vector.
\(M\) - the new master vector.

\textbf{Method:} The new version number \(X\) of \(P\) is set to equal the maximum of all \(X_i\)'s, \(i = 1, \ldots, k\). The new version vector \(V\) can be constructed as follows. For each \(v \in V\), do the following: if \(S_i[1] = 1\), set \(V[1] = 1\); otherwise, set \(V[1] = 0\) to equal the maximum of all \(V_i[1]\)'s, \(i = 1, \ldots, k\). The new master vector \(M\) is formed by 

\begin{verbatim}
procedure \textsc{resolve}(\(P\), \(V\), \(M\))
begin
  \(X \leftarrow \max(\{X_i| i = 1, \ldots, k\})\);
  for each \(v \in V\) do
    \(T \leftarrow \max(\{V_i[1] | j = 1, \ldots, k\})\);
    if there exists a \(v \neq T\) such that \(v = 0\) then \(V[1] \leftarrow 0\) else \(V[1] \leftarrow \max(\{v | v \in T\})\);
  end.
end.
\end{verbatim}

Fig. 4. Definition of procedure \textsc{resolve}.

protocol). When a copy of a file is updated, its version number \(X\) is incremented by 1. If \(\text{ismajority}\) returns no, the update access is rejected. However, the read-only accesses are allowed to proceed on all sites. Procedure \(\text{ismajority}\) is given in Figure 2.

When a site detects a network partitioning, it calls procedure \textsc{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 \(A\)). Figure 3 gives the definition of procedure \textsc{partition}.

When we merge two or more partitions into a single partition, we need to \textsc{resolve} and update the version numbers, the version vectors, and the master vectors of these copies. Assume \(k\) partitions \(P_1, P_2, \ldots, P_k\) are to be merged to form a new partition \(P\). The version number, the version vector, and the master vector of \(P\) are \(X, V, \text{and} M\), respectively. Procedure \textsc{resolve} is defined in Figure 4. The Example 2 illustrates how procedure \textsc{resolve} works.

\section{A Dynamic Majority Determination Algorithm}

\(\langle 8, 8, 0, 8 \rangle\), respectively. Resolving these two version vectors gives a new version vector \(V = \langle 0, 0, 0, 10 \rangle\). Sites \(A, B, C\) and \(D\) 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.

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.

Procedure \textsc{merge} (invoked at the site that initiates a merge)

\textbf{Input:} \(P\), \(P_2, \ldots, P_k\) - partitions to be merged (each \(P_i\) is a site in site name).

\textbf{Output:} \(P\) - a partition which is the merger of \(P_1, P_2, \ldots, P_k\).

\textbf{Method:} For each replicated file \(f\), do the following: first, request a site in each partition \(P_i\) to send the \(X_i, V_i, \text{and} M_i\) of its copy of \(f\). After having received all these \(X_i, V_i, \text{and} M_i\), invoke \textsc{resolve} to resolve them. Next, make all copies in \(P\) identical to partition \(X\). If \(P\) is now the majority partition, then attempt 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 \(X, V, \text{and} M\) to all the sites in \(P\) that have a copy of \(f\).

\begin{verbatim}
procedure \textsc{merge}(\(P\), \(P_2, \ldots, P_k\))
begin
  for each replicated file \(f\) do
    if there is a site \(a \in P\) that contains a copy of \(f\) then request a site in each partition \(P_i\) to send
      \(X_i, V_i, \text{and} M_i\) of its copy of \(f\); receive and store them in \(X_i, V_i, \text{and} M_i\);
    if there is no site in \(P\) that has a copy of \(f\) then give \(X_i, V_i, \text{and} M_i\) dummy values \(*
      \(*X_i \leftarrow 0; V_i \leftarrow <0,0,0,0>; M_i \leftarrow \langle 0,0,0,0,0\rangle\)
    end.
  end.
end.
\end{verbatim}

Fig. 5. Definition of procedure \textsc{merge}.
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 $\text{ISMAJORITY}$. 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 $\text{MERGE}$ in Figure 5 performs all merge operations mentioned above.

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 Figure 1. Assume the linear order of $f$ is such that $B > A > C$ (let $S = \langle A, B, C \rangle$ and $L = \langle 2, 3, 1 \rangle$). Suppose that after $f$ is created, two updates are successfully performed on $f$. The current state of $f$ is as follows:

\[
\begin{align*}
A, B, C \\
X &= 2 \\
V &= \langle 0, 0, 0 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]

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 $\text{PARTITION}$, we have the following state:

\[
\begin{align*}
A \\
X &= 2 \\
V &= \langle 0, 2, 2 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]
\[
\begin{align*}
B, C \\
X &= 2 \\
V &= \langle 2, 0, 0 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]

At this moment, if site $A$ invokes $\text{ISMAJORITY}$, then Set1 will be set to $\{ A \}$ and Set2 will be set $\{ B, C \}$. If $B$ or $C$ invokes $\text{ISMAJORITY}$, 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:

\[
\begin{align*}
A \\
X &= 2 \\
V &= \langle 0, 2, 2 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]
\[
\begin{align*}
B \\
X &= 5 \\
V &= \langle 2, 0, 5 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]
\[
\begin{align*}
C \\
X &= 5 \\
V &= \langle 2, 5, 0 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]

Both partition $B$ and partition $C$ contain one-half of the current copies ($|\text{Set1}| = |\text{Set2}| = 1$ if $\text{ISMAJORITY}$ 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 of copies of that file.

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

\[
\begin{align*}
A, C \\
X &= 5 \\
V &= \langle 0, 5, 0 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]
\[
\begin{align*}
B \\
X &= 8 \\
V &= \langle 2, 0, 0 \rangle \\
M &= \langle F, F, F \rangle
\end{align*}
\]

Note that $A$'s marker in partition $AC$ has been changed to $T$ (marked). If $A$ or $C$ later invokes $\text{ISMAJORITY}$, 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 create a cycle in the conflict graph [14].
2. The read-only accesses considered one at a time in the conflict graph do not create 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 want 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 one's 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 relays due to a failure of some part of the system or network partition. Of course, when one 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 merging 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:

$$\begin{align*}
A, C & \quad B \\
X &= 5 & X &= 8 \\
V &= \langle 0, 5, 0 \rangle & V &= \langle 2, 0, 5 \rangle
\end{align*}$$

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 $C$, the merge 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 a restriction might become quite severe and unacceptable, as Example 4 illustrates.

**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 Figure 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 are allowed. They are: $A$ merges with $B$; and $C$ merges with

\begin{enumerate}[$D]$ 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 nonmajority partition, mentioned at the end of the 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 the majority partition is unavoidable. We say that a partitioning is simple if it splits a partition into exactly two partitions. Otherwise, it is called multiple. A multiple partitioning might cause a file to be inaccessible everywhere, as the partition graph in Figure 7 illustrates. In the class of dynamic voting algorithms, no solution exists for such a 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, must be made. No partition will claim itself the majority partition.
A DYNAMIC MAJORITY DETERMINATION ALGORITHM

The reader might notice that, in our algorithm, if the majority partition is lost (due to the occurrence 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

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

As an example, for the history in Figure 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

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

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

$$A^*_j = \frac{3 + 3 + 4 + 3 + 3}{100} = \frac{16}{100}.$$ 

The first method works better in this case.

We are investigating a design of an efficient scheme that can adapt to a
changing environment (e.g., configuration of the network) by selecting appro-
priate algorithms to allow high availability. Experiments on the RAID system
[6] 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
2. D. Barbara, H. Garcia-Molina, and A. Spaulster, Protocols for Dynamic Vote Reassign-
3. F. A. Bernstein and N. Goodman, Concurrency control in distributed database systems,
4. B. Bhargava, Transaction processing and consistency control of replicated copies during
5. B. Bhargava, Tie Declaration and a New Majority Establishment during Network Recon-
figuration, working paper, Computer Science Dept., Purdue Univ., May 1987.
6. B. Bhargava, J. Reidl, and A. Royappa, The RAID Distributed Database System, TR 691,
7. D. Davison and W. Burkhard, Consistency and recovery control for replicated files, in
Proceedings of the 10th ACM Symposium on Operating Systems Principles, Orcas Island,
8. S. B. Davidson, H. Garcia-Molina, and D. Skeen, Consistency in partitioned networks,
9. C. Ellis, A robust algorithm for updating duplicated databases, in Proceedings of the
Second Berkeley Workshop on Distributed Management of Data and Computer Networks,
10. H. Garcia-Molina, Performance of Update Algorithm for Replicated Data in a Distributed
11. S. Jajodia, Managing replicated files in partitioned distributed database systems, presented
12. S. Jajodia and D. Muchler, Dynamic voting, presented at ACM SIGMOD-87, San
13. S. Jajodia, and D. Muchler, Enhancements to the voting algorithm, in VLDB, Brighton,
16. J. A. Stankovic, K. Ramamritham, and W. H. Kohler, A review of current research and
critical issues in distributed system software, in Concurrency and Reliability in Distributed
17. M. Stonebraker, Concurrency control and consistency of multiple copies of data in
18. R. Thomas, A majority consensus approach to concurrency control, ACM Trans.
1978).

Received 10 September 1987