# A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases

ROBERT H. THOMAS
Bolt Beranek and Newman, Inc.

A "majority consensus" algorithm which represents a new solution to the update synchronization problem for multiple copy databases is presented. The algorithm embodies distributed control and can function effectively in the presence of communication and database site outages. The correctness of the algorithm is demonstrated and the cost of using it is analyzed. Several examples that illustrate aspects of the algorithm operation are included in the Appendix.

Key Words and Phrases: distributed databases, distributed computation, distributed control, computer networks, update synchronization, concurrency control, clock synchronization, multiprocess systems
CR Categories: 4.3, 4.32, 4.33, 4.39, 5.29

## 1. INTRODUCTION

In a computer network environment it is often desirable to store copies of the same database at a number of different network sites. A number of advantages can result from maintaining such duplicate databases. Among these advantages are: increased data accessibility—the data may be accessed even when some of the sites where it is stored have failed, as long as at least one of the sites is operational; more responsive data access—database queries initiated at sites where the data are stored can be satisfied directly without incurring network transmission delays and those initiated from sites "near" the database sites can be satisfied with less delay than those "farther" from the database sites; load sharing—the computational load of responding to queries can be distributed among a number of database sites rather than centralized at a single site.

These and other benefits of replicating data must be balanced against the additional cost and complexities introduced in doing so. There is, of course, the cost of the extra storage required for the redundant copies. This paper considers the problem of maintaining synchronization of multiple copy databases in the presence of update activity and presents a solution to that problem. Other problems (e.g. determining for a given application the number of copies to

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N0014-75-0773.

Author's address: Bolt Beranek and Newman, Inc., 50 Moulton St., Cambridge, MA 02138.

maintain and the sites at which to maintain them; selecting a database site or sites to satisfy a query request when it is initiated) are not considered in this paper.

The inherent communication delay between sites that maintain copies of a database makes it impossible to ensure that all copies remain identical at all times when update requests are being processed. The principal goal of an update mechanism is to guarantee that updates get applied to the database copies in a way that preserves the *mutual consistency* of the collection of database copies as well as the *internal consistency* of each database copy. By mutual consistency we mean that all copies converge to the same state and would be identical should update activity cease. The notion of internal consistency is somewhat more difficult to define precisely. It concerns the preservation of invariant relations that exist among items within a database. As such, internal consistency is related to the interpretation or semantics of items in the database. Therefore, most of the responsibility for the internal consistency of a database must rest with the application processes which update it. The update mechanism should incorporate little, if any, knowledge of the database semantics. It should, however, operate in a manner that does not destroy internal data relationships if the application processes updating the database act in a way that preserves them.

An example should serve to clarify the distinction between the two types of consistencies. Consider a simple database duplicated at sites $A$ and $B$ that includes data items $x$, $y$, and $z$, which all initially have the value 1 in both copies. Further assume that the relation

$$x + y + z = 3$$

must be preserved for the database. Consider two updates $R1$ and $R2$:

$$\begin{aligned} R1&: x := -1, \quad y := 3 \\ R2&: y := -1, \quad z := 3 \end{aligned}$$

each of which, based on the initial database state, preserves the relation. If $R1$ and $R2$ are both applied, regardless of the order of application, the internal consistency of the database (the relation $x + y + z = 3$) will be destroyed. Hence (at least) one of the requests must be rejected in order to preserve the internal consistency of the database. Stated somewhat differently, the update request that gets rejected must be refused because it is based on information made obsolete (the initial values of $x$, $y$, and $z$) by the request that gets accepted. Concurrency control mechanisms designed to maintain internal consistency for single copy databases typically use some sort of mutual exclusion scheme (lock or semaphore discipline) to guarantee that updates applied are based on current information. The mutual consistency of the database would be destroyed, while its internal consistency would be preserved, if update $R1$ was accepted at site $A$ and update $R2$ was accepted at site $B$. Maintenance of mutual consistency requires all sites to make the same decision for concurrently initiated conflicting updates.

Update mechanisms can be characterized in terms of the control disciplines they utilize. One class of mechanisms involves some form of centralized control whereby all update requests pass through a single central point. At that point the requests can be validated and then distributed to the various database sites for application to the database copies. A second, fundamentally different class of update mechanisms embody distributed control. For this class the responsibility for validating update requests and applying them to the database copies is distributed among the collection of database sites.

Mechanisms which use centralized control are attractive because a central control point makes it relatively easy to detect and resolve conflicts between update requests which, if left unresolved, could lead to inconsistencies and eventual divergence of the database copies. The primary disadvantage of such mechanisms is that database update activity must be suspended whenever the central control point is inaccessible. Such inaccessibility could result from failures in the communications network or at the site where the control point resides. Because a distributed control update mechanism has no single point of control, it should, in principle at least, be possible to construct one capable of processing database updates even when one or more of the component sites are inaccessible.[1] The problem here is that it is nontrivial to design a distributed update control mechanism which operates correctly; that is, which can resolve conflicting updates in a way that preserves consistency and is deadlock-free. Centralized update control is adequate for many applications; however, there are database applications whose update performance requirements can be satisfied only by a system which uses distributed update control.

This paper presents an algorithm for maintaining multiple copy databases which uses distributed control. The algorithm treats all database sites as more or less equivalent. For example, an update can be initiated at any site.

The algorithm presented can be characterized as a *majority consensus* algorithm. Database sites vote on the acceptability of update requests. For a request to be accepted and applied to all database copies only a majority need approve it. The voting procedure followed by each database site allows it to approve an update request only if the information upon which the request was based is valid when it votes. The algorithm employs a timestamping mechanism used both in the voting procedure and in the application of accepted updates to the database copies.

The update algorithm can be demonstrated correct in the sense that its operation is deadlock-free and preserves both internal consistency and mutual consistency of the database copies. Although a formal correctness proof of the algorithm is beyond the scope of this paper, strong plausibility arguments for its correctness are presented.

An important property of the algorithm is its ability to recover from and function effectively in the presence of communication system and database site failures. For example, the algorithm is robust with respect to lost and duplicate messages and the temporary inability of database managing processes to communicate with one another. In addition, it can be made resilient to the loss of memory (state information) by one or more of the database managing processes. Unlike many other update algorithms [1-3], the robust behavior of the majority consensus algorithm does not require the database system to detect component

[1] A "distributed" mechanism that comes quickly to mind is one which locks all copies of the database for the duration of the update activity. Since such a mechanism requires every database site to be accessible to process an update, it is even more vulnerable to component outages than one which uses centralized control.

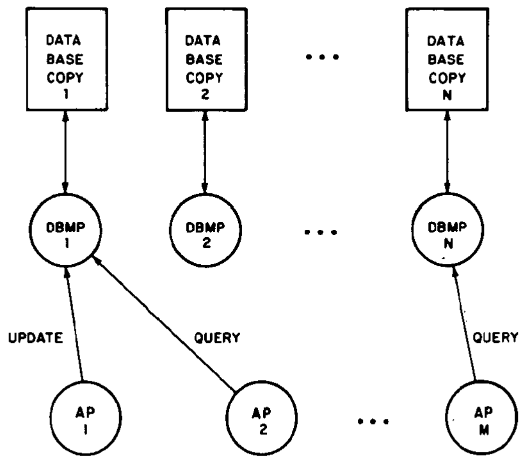

Fig. 1. Each database copy is directly accessible only through its local database managing process (DBMP) which acts on behalf of application processes (APs) to process their query and update requests

malfunctions or outages and to reconfigure or switch into a special recovery mode of operation. Rather, its robustness is achieved as a side effect of its normal operation.

## 2. DISTRIBUTED DATABASE ENVIRONMENT

We assume an environment within which copies of a database are accessible at a number of database sites (see Figure 1). As presented, the algorithm assumes full replication of the database at all sites. The algorithm can be adapted to deal with partially replicated data, but this paper does not discuss partial data replication further. It is further assumed that the database copy at each site is accessible only through a database managing process (DBMP) which resides at that site. Query and update accesses to the database are initiated by application processes (APs). Each access to the database is completed by a DBMP acting on behalf of the initiating AP.

The database is assumed to consist of a collection of named elements. For our purposes the nature of such an element is unimportant. An element could be a record, a field within a record, a collection of records, or a simple variable as in the example above. Each named element has a value and a timestamp associated with it. An element's timestamp represents the time that the element received its current value. As will be shown, timestamps are used in two ways. They are used during update synchronization to ensure the preservation of internal database consistency, and perform a function similar to that of mutual exclusion locks. In addition, timestamps are used in the procedure followed by a DBMP when it applies an update to its database copy. This "update application" procedure is designed to guarantee mutual consistency of the database copies. Just as lock granularity is an implementation issue in systems that use locks for concurrency control, the granularity of timestamps is an issue for systems using majority consensus. For the purpose of presenting the majority consensus algorithm it is useful to think of a timestamp being associated with the smallest separately modifiable database element. However, for storage efficiency an implementer might choose to have collections of related elements share timestamps.

To query the database an AP sends a query request to a DBMP. The DBMP acts upon the request by querying its copy of the database and returning the results to the requesting AP.

The manner in which updates are performed is somewhat more involved. We assume that, in general, an AP initiates an update by first performing a computation to generate new values for certain database elements using database values obtained by one or more queries, and then submitting an update request to a DBMP which cooperates with the other DBMPs to perform the update. The update procedure can be decomposed into the following sequence of steps.

(1) *Query Database.* The AP queries the database to obtain data element values to use in its update computation. The set of data elements used by the AP are called the *base variables.* In addition to their values, the DBMP responding to the query also supplies the base variable timestamps stored in its copy of the database.
(2) *Compute Update.* The AP computes new values for the data elements to be updated. The set of data elements to be updated are called the *update variables.* The algorithm requires that the update variables be a subset of the base variables.
(3) *Submit Request.* The AP constructs an *update request* composed of the update variables with their new values and the base variables with their timestamps, and submits it to a DBMP.
(4) *Synchronize Update.* The DBMP set cooperates to decide to accept or reject the request. Each DBMP participating in the decision executes the same voting procedure for the request. The voting procedure itself involves checking the base variable timestamps in the update request against the corresponding timestamps in the local copy of the database. This check allows the DBMP to determine whether the base variables have been modified since their values were obtained in the query step.
(5) *Apply Update.* If the request is accepted, each DBMP applies the update to its copy of the database.
(6) *Notify AP.* A DBMP informs the AP how the request was resolved. If it was rejected, the AP may resubmit it by repeating this sequence of steps.

This update procedure involves two sorts of interprocess communication. Communication between APs and DBMPs occurs to initiate updates and report their outcome. The second kind of communication occurs between DBMPs as they work to reach a consensus on the outcome of update requests and to ensure that updates accepted get applied to all database copies.

For purposes of this paper we assume the existence of a suitable interprocess communication facility. We further assume that the communication system is reliable in the sense that the delivery of interprocess messages is guaranteed even if the receiving process is inaccessible when the sending process initiates a message transmission. Implementation of such a facility usually involves persistence, until receipt of a positive acknowledgment indicating that the message was successfully delivered.[2]

## 3. THE MAJORITY CONSENSUS ALGORITHM

The majority consensus algorithm consists of five rules.

(1) *DBMP/DBMP Communication Rule.* This rule defines the communication patterns used by the DBMP set as it cooperates to arrive at a consensus on update requests. Possible communication patterns include daisy chaining, daisy chaining with timeouts and retransmission (see below), and broadcasting.
(2) *Voting Rule.* This is the rule followed by each DBMP when it considers an update request. Details of the voting rule are sensitive to the DBMP/DBMP communication rule.
(3) *Request Resolution Rule.* After DBMP voting this rule is applied to determine the outcome of the voting. Its details depend upon the particular DBMP/DBMP communication rule used.
(4) *Update Application Rule.* This rule governs the way updates are entered into the database copies. When a DBMP learns that an update request has been accepted, it uses the update application rule to perform the update on its database copy.
(5) *Timestamp Generation Rule.* A timestamp is generated and assigned to each update request initiated by an AP. When an accepted update is applied to a database copy, values for the data elements which are update variables are modified as specified and the timestamps in the database copy for the modified database elements are set to the timestamp assigned to the update.

The voting and request resolution rules constrain DBMP behavior such that no two "conflicting" concurrent update requests can be accepted. This guarantees that the update application rule, which operates to ensure mutual consistency, can be used safely without destroying internal database consistency. The rest of this section describes these rules in more detail.

### 3.1 DBMP/DBMP Communication Rule

Update requests made by APs must be communicated among the DBMPs for voting and DBMP votes must be communicated to be tallied. Two possible communication disciplines are (see Figure 2):

(1) *Broadcast.* The DBMP receiving an AP update request broadcasts it to the other DBMPs for voting. After voting the DBMPs return their votes to the original DBMP (and perhaps to each other if highly failure tolerant operation is required) for request resolution.
(2) *Daisy Chain.* The DBMP receiving the request votes and forwards the request along with its vote to another DBMP. That DBMP, in turn, votes

[2] The "network mail" facility of the ARPANET [4] incorporates such a reliable transmission mechanism which ensures that network mail is always eventually delivered. The details of how such mechanisms can be implemented, though important, will not be discussed further here.

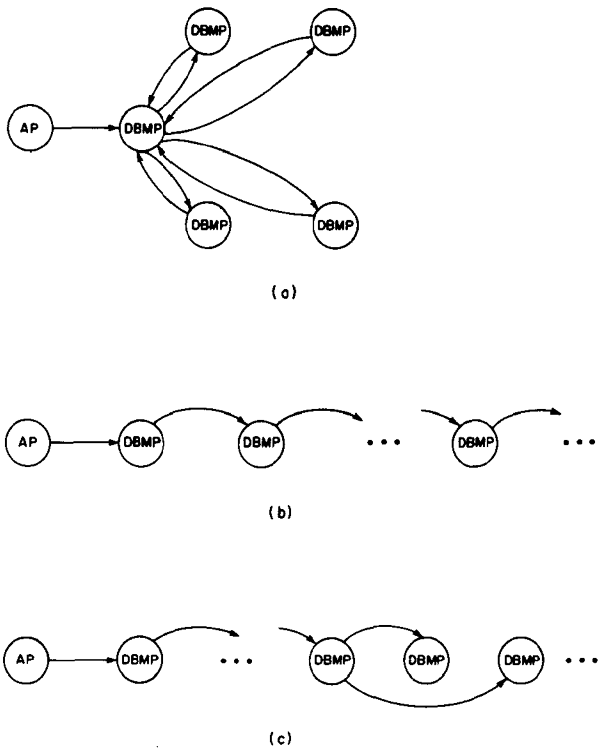

Fig. 2. Possible DBMP/DBMP communication patterns include: (a) broadcasting; (b) daisy chaining; and (c) daisy chaining with timeout and retransmission

and forwards the request and the votes along to another DBMP that has not voted yet. This procedure continues until the request is resolved.

Use of a broadcast discipline allows requests to be resolved with minimal delay at the possible expense of extra messages.[3] Daisy chaining results in resolution with the minimum number of messages at the expense of relatively high delay. In practice, the choice of a communication discipline should be based upon performance requirements for the database system as well as the characteristics of the underlying communication system.[4]

The details of the voting and request resolution rules are sensitive to the DBMP/DBMP communication rule and are presented below assuming daisy chaining and the following variant of it which is introduced as a means for increasing the robustness of the algorithm.

With pure daisy chaining, progress toward the resolution of an update request can temporarily cease only if: (1) a DBMP trying to forward an unresolved request is unable to find another DBMP that is accessible and has not yet voted

[3] Since only a majority concensus is required, any messages to solicit and tally votes in excess of the majority are extra.

[4] For example, in a computer network such as the ARPANET [5], broadcasting is relatively expensive, whereas it is relatively inexpensive in a satellite-based communication system or a local computer network such as an Ethernet [6].

on the request; (2) a DBMP trying to forward an unresolved request crashes before fowarding the request.

In the first case there is little to be done until a DBMP that has not already voted becomes accessible. We assume that the sending DBMP is persistent and will forward the request when a nonvoting DBMP becomes accessible.

Timeouts can be used to detect and recover from failures of the second type. A DBMP which has voted and forwarded an unresolved update request could time the request out and if it does not learn that the request has been resolved within the timeout period, it could act to help the request progress further toward resolution.

A procedure that a DBMP can use when a request has been timed out is to check with the DBMP ($X$) to which it forwarded the request. If it cannot communicate with $X$, then it should attempt to forward the update request to some other DBMP that, to its knowledge, has not yet voted. If $X$ is up and knows about the request, the checking DBMP need only reactivate the request timeout since it can assume that $X$ is using the same procedure to ensure that the request proceeds toward resolution.[5]

This procedure is analogous to the "timeout and retransmit" procedures used in many network communication protocols [7]. It contributes to the robustness of the database update algorithm by insuring that the system of DBMPs works toward the resolution of an update request as long as at least one DBMP which knows about the request is functioning.

## 3.2 DBMP Voting Rule

The voting rule is the basis for concurrency control. Together with the request resolution rule it insures that mutual exclusion is achieved for possibly conflicting concurrent updates. Two requests are said to *conflict* if the intersection of the base variables of one request and the update variables of the other request is not empty.

The basic idea of the voting rule is simple. When considering an update request a DBMP checks to determine whether any of its base variables have been modified since the request was constructed by the initiating AP. If none have been changed, then so far as the DBMP can determine, the premises upon which the update is based remain valid since no conflicting updates have been accepted since the request was initiated or are currently in progress (see below). Consequently, the DBMP may vote (OK) to accept the request. If any base variable has been modified, then the premises are no longer valid and the DBMP must vote (REJ) to reject the request. The currency of the request base variables is checked by comparing their timestamps, which are supplied as part of the request, with the corresponding timestamps stored in the DBMP's database copy. Between the time a DBMP votes on a request and the request is resolved, the request is said to be *pending* at that DBMP.

Two factors contribute to complicate the voting rule somewhat. First, from time to time a DBMP must consider a request that conflicts with a pending

[5] A DBMP might choose the timeout period to be a function of the number of votes a request has accumulated to account for the fact that, in most cases, it will take a request with few votes relatively longer to be resolved than one with many votes.

request. This typically occurs when two APs concurrently initiate conflicting requests. Even if the base variables of the second request are current, the DBMP is not free to vote OK since the pending request may be accepted and render the base variables for the second request obsolete. One possibility would be for the DBMP to defer voting on the second request until the pending conflicting request is resolved. The problem with deferring in this way is that it introduces the second complicating factor: the possibility of deadlock. For example, for two requests initiated at different DBMPs in a two DBMP system, voting on each could be deferred at one of the DBMPs pending resolution of the other.

The solution to this problem as embodied in the voting rules stated below is to introduce a new vote. A PASS vote is made when it is necessary to signal other DBMPs that a potential deadlock situation exists with respect to the request in question. When voting on a request that conflicts with a pending request, a DBMP must either vote PASS or defer voting until the conflicting request is resolved. The choice made by the DBMP depends upon the relation of the requests in question to one another. The rule as stated below uses a *priority* scheme: each request is assigned a priority and the priority of conflicting requests are compared when the DBMP votes. A simple priority scheme is used where the priority of a request is the timestamp assigned to it. Timestamps produced by the timestamp generation rule will be shown in Section 3.5 to be unique.

The DBMP *voting rule* can now be stated:

(1) Compare the timestamps for the request base variables with the corresponding timestamps in the local database copy.
(2) Vote REJ if any base variable is obsolete.
(3) Vote OK and mark the request as pending if each base variable is current and the request does not conflict with any pending requests.
(4) Vote PASS if each base variable is current but the request conflicts with a pending request of higher priority.
(5) Otherwise, defer voting and remember the request for later reconsideration.

Voting will be deferred if the request conflicts with a pending request of lower priority or if any base variable is more current than the corresponding data element in the database. Some request base variables could be more current if the update request was initiated at a DBMP which had previously applied an update that changed them and about which the voting DBMP has not yet learned.

If the timeout and retransmit communication discipline is used, it is possible for a DBMP to be asked to vote on a request for which it has already voted. A DBMP is forbidden from changing its vote in such a case.

## 3.3 Request Resolution Rule

After voting a DBMP uses the request resolution rule to check whether its vote resolved the request. The basic idea is that the request should be accepted if a majority of the DBMPs have voted OK. The important property of majority consensus is that the intersection of any two majorities has at least one DBMP in common.[6] This means that for any two requests that are accepted at least one DBMP voted OK for both.

The DBMP *request resolution rule* for daisy chaining has two parts:

(1) After voting on a request ($R$):
  (a) if the vote was OK and a majority consensus exists:
  accept $R$ and notify all DBMPs and the AP that $R$ was accepted.
  (b) if the vote was REJ:
  reject $R$ and notify all DBMPs and the AP that $R$ was rejected.
  (c) if the vote was PASS and a majority consensus is no longer possible (see discussion below):
  reject $R$ and notify all DBMPs and the AP that $R$ was rejected.
  (d) otherwise, forward $R$ and the votes accumulated so far to a DBMP that has not voted on it.

(2) After learning that a request ($R$) has been resolved:
  (a) if $R$ was accepted:
    (i) Apply $R$ to the local copy of the database using the update application rule (see Section 3.4).
    (ii) Reject conflicting requests that were deferred because of $R$.
  (b) if $R$ was rejected:
    (i) Use the voting rule to reconsider conflicting requests that were deferred because of $R$.

Part 1(c) of the rule prevents deadlocks. Recall that a DBMP votes PASS only when the request base variables conflict with those of another pending request in order to inform other DBMPs that a potential deadlock situation with respect to the request exists. The request in question can continue to be considered by other DBMPs until sufficient PASS votes accumulate to prevent a majority consensus. If this condition occurs, the DBMP detecting it must reject the request. In effect, this condition represents a consensus among the DBMPs that the request should be rejected to prevent a possible deadlock. To see the kind of situation rule 1(c) prevents, consider a $2N$ DBMP system for which two conflicting update requests are initiated at different DBMPs. Each request could progress to the point where each has $N$ OK votes. Without a rule such as the PASS rejection rule, neither could achieve a majority consensus and a deadlock would result.

Note that for the daisy chain communication discipline a single dissenting vote is sufficient to cause a request to be rejected. If daisy chaining with timeout and retransmit is used, the interpretation of a REJ vote must be weakened as follows:

(1) (b) if the vote was REJ and a majority consensus is no longer possible: reject $R$ and notify all DBMPs and the AP that $R$ was rejected.

The reason for weakening REJ is to prevent the DBMP set from both accepting and rejecting the same request. With pure daisy chaining a strong REJ (veto)

[6] A natural way to define majority subsets for an $N$ DBMP set is any collection of $N/2 + 1$ DBMPs. Other groups of majority subsets are possible, however. The only requirement is that any two have a nonempty intersection. It might be advantageous to include a particular DBMP in most or every majority subset if, for example, it was resident on a very fast or highly reliable system.

works because a request follows a single path through the DBMP set as it proceeds toward resolution and only one DBMP, the last one that votes, decides the fate of the request. If daisy chaining with timeout and retransmission is used, it is possible for the request's path to branch into several paths,[7] possibly allowing several DBMPs to decide the fate of the request. Weakening the REJ vote ensures that each DBMP makes the same decision.

## 3.4 Update Application Rule

Due to the way updates are accepted by and communicated among the DBMP set it is possible for notification of the acceptance of an update ($R1$) to arrive at some DBMP after notification of the acceptance of a later update ($R2$) which obsoletes $R1$.

For example, consider a 3 DBMP system where DBMP 3 is down when $R1$ and $R2$ are accepted. Further, suppose that $R1$ was initiated by an AP at DBMP 1 and accepted at DBMP 2, and that $R2$ was later initiated at DBMP 2 and accepted at DBMP 1. Now, assume that when DBMP 3 comes up DBMP 2 is down. DBMP 3 will receive notification of $R2$'s acceptance from DBMP 1; sometime later, when DBMP 2 comes up, DBMP 3 will receive notification of $R1$'s acceptance from DBMP 2. It is important that any data elements updated by both $R1$ and $R2$ have the values assigned by $R2$ after DBMP 3 has applied both updates.

The *update application rule* ensures that accepted updates are "properly sequenced" as they are incorporated into the database copies:

To apply an update ($R$) to a variable ($v$) in a database copy, compare the timestamp of $R$ ($T$) with the timestamp of the variable in the database ($Tv$): If $Tv < T$, modify $v$ and set $Tv$ to $T$; otherwise, omit the update to $v$ since it is obsolete.

For updates with more than a single update variable it may be the case that some of the assignments to data elements are performed and others are omitted as obsolete.[8]

## 3.5 Timestamp Generation Rule

Timestamps are used in two ways. They are used in the voting rule to determine the currency of update request base variables and they are used in the update application rule to guarantee that recent updates supersede older ones. In both

[7] A path can branch if a DBMP that failed and was "bypassed" after a timeout is later restarted and resumes normal processing of the request. Refer to Figure 2(c).

[8] Since updates are applied in the order in which notification of their acceptance arrives at a DBMP, it is possible that some DBMP $X$ will apply them in a different order than they were accepted. Consequently, it is possible for the internal consistency of the database copy at $X$ to be temporarily destroyed. However, the consistency will be restored when the "missing" updates are applied at $X$. Furthermore, it can be shown that any updates initiated at $X$ that are based on the inconsistent data will be rejected by the DBMP set. If an application requires that such temporary inconsistencies never occur, it is possible to reformulate the update application rule such that an update will be applied to a database copy only after all previously accepted updates have been applied. Implementation of such a modified update application rule requires more communication among the DBMPs regarding updates that have been accepted.

cases timestamps are used as sequence numbers for ordering events (e.g. conflicting update requests) with respect to one another.[9]

The properties that the timestamps used in this way should have are: uniqueness (i.e. no two update requests should have the same timestamp); and monotonicity (i.e. successively generated timestamps should increase).

All timestamps in the DBMP system ultimately come from update requests. The question of when and by whom the requests should be timestamped arises. There seem to be only two logical choices: by the initiating DBMP at the time the update is received from an AP; or by the accepting DBMP at the time the update is accepted.

The timeout and retransmit communication discipline makes it possible for a given request to be accepted by more than a single DBMP. Therefore, to ensure a single, unique timestamp, requests are timestamped by the DBMP with which the request is initiated.

It is assumed that each DBMP has access to a local, monotonically increasing clock, but that there is no common clock accessible to all DBMPs. A *timestamp* generated by a DBMP $i$ is a pair $(T, i)$ where $T$ is the time obtained from the local DBMP clock. $T$ is called the *c-part* (for clock) of the timestamp and $i$ the *d-part* (for DBMP) of the timestamp.

Equality, greater than, and less than for timestamps can be defined as follows.

Let $T1 = (C1, d1)$ and $T2 = (C2, d2)$.
Equality (=): $T1 = T2$ if and only if $C1 = C2$ and $d1 = d2$.
Greater than (>): $T1 > T2$ if and only if $C1 > C2$ or $C1 = C2$ and $d1 > d2$.
Less than (<): $T1 < T2$ if and only if $C1 < C2$ or $C1 = C2$ and $d1 < d2$.

It is not difficult to ensure that timestamps are unique. A DBMP need only take care to never assign the same $c$-part to timestamps for different update requests.

The possibility that the local DBMP clocks are skewed with respect to one another or run at different rates could lead to certain anomalous behavior [8]. In terms of the example in Section 3.4, anomalous behavior would result if the timestamp generated by DBMP 1 for $R1$ is more recent than that generated for $R2$ by DBMP 2; the anomaly here would be that $R1$, the earlier update, would be retained in the database. We shall call such an occurrence a *sequencing anomaly*. If permitted to occur, sequencing anomalies could destroy the internal consistency of the database. For example, consider a database with the internal consistency requirement that $x + y + z = 3$ for which initially $x = 2$, $y = 0$, and $z = 1$ where

$$\begin{aligned} R1&: x := 0, \quad y := 2 \\ R2&: x := 1, \quad z := 0. \end{aligned}$$

Each update preserves the internal consistency requirement (assuming as in Section 3.4 that $R2$ is initiated after $R1$ is accepted). However, if a sequencing anomaly were to occur, the internal consistency requirement would be violated after $R1$ and $R2$ were applied since $x + y + z = 2$.

[9] For purposes of the algorithm, sequence numbers would suffice. However, we shall continue to use timestamps rather than sequence numbers because of their simple intuitive appeal and because the date and time information they carry to support event ordering is useful in its own right.

The *timestamp generation rule* used by a DBMP $i$ to assign a timestamp to a request $R$ is:

Let

$$T = 1 + \max(\text{time}, \max(\{Tb\}))$$

where "time" is the time obtained by DBMP $i$ from its local clock and the $\{Tb\}$ are the $c$-parts of the timestamps for $R$'s base variables. The timestamp for $R$ is $(T, i)$. To ensure that its local clock increases and that it never regenerates the same timestamp, DBMP $i$ also resets its local clock to $T$.

It will be shown in Section 4 that this rule prevents the occurrence of sequencing ing anomalies.

## 3.6 Discussion

Several observations can be made regarding the robustness of the majority consensus algorithm. Only a majority of the DBMPs or fewer (in the case of a rejection for pure daisy chaining) are necessary for an update request to be resolved. Therefore, the database can undergo modification when some of the DBMPs are inaccessible. Furthermore, the DBMPs necessary for a majority consensus need not be simultaneously available. Since the algorithm involves only pairwise interactions among DBMPs, an update request can advance toward resolution when only two DBMPs are up.

It is not necessary for the DBMP where a request is initiated to remain up for the request to be resolved. The initiating DBMP need only remain active sufficiently long to vote on the request and forward it to another DBMP. The requesting AP is notified by the DBMP that detects resolution of the request, rather than by the initiating DBMP.

The timeout and retransmit discipline provides an additional degree of robustness. This additional robustness comes at an easily identifiable cost: the meaning of the REJ vote must be weakened to achieve it. As a consequence more votes, and therefore additional delay and messages, are required to reject an update.

A side effect of retransmission is that a DBMP may be asked to consider a given update request more than once. This is analogous to the receipt of duplicate messages in a communication system which uses retransmission. Duplicate requests represent no problem as long as DBMPs can detect them and do not change their votes.

When voting on an update request, a DBMP must be able to determine whether it has previously voted on the request. Similarly, in order to be able to "garbage collect" storage used for maintaining state information for pending requests, when informed of the resolution of a request, a DBMP must be able to determine whether it has any record of the request.[10]

It follows that update requests must be uniquely identified within the set of DBMPs. We note that the update timestamp generated by the initiating DBMP for a request is unique and, therefore, is adequate to serve as a unique request identifier.

[10] A DBMP may "discard" an accepted update request after it has entered the update into its database copy. DBMPs also "discard" rejected requests. This paper does not discuss how a DBMP can tell when it is safe to discard a request; however, it is not difficult to devise methods for doing so.

The Appendix to this paper presents several examples which illustrate how update requests submitted to a DBMP set proceed toward resolution under the communication, voting, and request resolution rules presented in the previous sections.

## 4. WHY THE ALGORITHM WORKS

The way the voting, request resolution, and update application rules make use of the properties of majority subsets and timestamps is the basis for the majority consensus algorithm.

The mutual exclusion necessary to make preservation of internal consistency possible is accomplished through the voting and request resolution rules which constrain individual DBMP behavior with respect to conflicting update requests. In particular, concurrency control is achieved as a consequence of the resolution rule that ensures that the consenting majority subsets for any two accepted requests have at least one DBMP in common, and the voting rule that prevents a given DBMP from consenting (voting OK) to two conflicting concurrent update requests.

The update application rule guarantees that the database copies are mutually consistent by ensuring that recent updates supersede older ones. In effect it enables each DBMP to reconstruct and act upon the same sequence of update events regardless of the order in which different DBMPs learn of different updates.

A formal proof of the correctness of the algorithm is beyond the scope of this paper; however, the rest of this section informally argues for its correctness. In particular, to establish the correctness of the algorithm we claim that:

(1) An update request $R$ is either accepted or rejected by the DBMP set but not both.
(2) The algorithm is deadlock-free.
(3) All copies of the database converge to the same value.
(4) The algorithm implements mutual exclusion for accepted updates, and therefore acts in a way that preserves internal database consistency.
(5) Sequencing anomalies cannot occur.

*Claim* 1. An update request $R$ is either accepted or rejected by the DBMP set but not both.

*Argument:* Case $A$ (pure daisy chaining—strong REJ). $R$ follows a nonbranching sequential path through the DBMP set. After voting on $R$, each DBMP along the path applies the request resolution rule to $R$ before forwarding it to another DBMP. If $R$ is resolved the path is terminated. Thus the resolution decision for $R$ is made by a single DBMP which is constrained by the request resolution rule to either accept or reject $R$.

*Argument:* Case $B$ (daisy chaining with timeout and retransmit—weak REJ). Because $R$ may follow a branching path through the DBMP set a weak REJ is used to ensure that if several DMBPs make the resolution decision for $R$ each makes the same decision. Assume $R$ is both accepted and rejected. To be accepted the request resolution rule requires that a majority of DBMPs voted OK on $R$; to be rejected it requires that enough DBMPs voted PASS or REJ $R$ to prevent a majority consensus. This can only occur if some DBMP voted both OK and

PASS/REJ. But the voting rule prohibits this. Therefore, $R$ cannot be both accepted and rejected.

*Claim* 2. An update request $R$ will be resolved by the DBMP set in finite time if given any pair of DBMPs in the set, they are capable of interacting with one another in a finite time. It follows therefore that the DBMP set is deadlock-free.

*Argument* (for pure daisy chaining). Let $R$ be initiated at DBMP $I$. $I$ has four options with respect to $R$: (1) it can vote REJ on $R$; (2) it can vote OK on $R$; (3) it can vote PASS on $R$; or (4) it can defer voting on $R$.

If DBMP $I$ votes REJ, $R$ is resolved in finite time. Consider cases (2) and (3). After voting, $I$ can forward $R$ to another DBMP $J$ that has not voted on $R$. The premise assures that this is done in finite time.

DBMP $J$ has the same four options with respect to $R$. If it rejects $R$, $R$ is resolved in finite time. If it votes OK or PASS and there are insufficient votes to resolve $R$, $J$ will forward $R$ to another DBMP $K$ that has not yet voted on $R$, thereby, in finite time, advancing $R$ one step closer toward resolution. Since there are at most $N$ (the number of DBMPs) such steps required to resolve $R$, it suffices to show that each step requires only finite time.

The only case that is potentially troublesome is when a DBMP defers voting. A DBMP will defer voting only if $R$ conflicts with a pending request of lower priority or if the timestamps for $R$'s base variables are too current. If it can be shown that voting on $R$ cannot be deferred indefinitely, then $R$ will either be resolved or advanced one step further toward resolution by the DBMP in finite time.

There can be at most only a finite number of requests with priority less than $R$'s. This is so since a request's priority is its timestamp, and due to the way timestamps are generated, there can be at most a finite number of requests with timestamps less than $R$'s. Consider first the case in which there is a request corresponding to every possible timestamp less than $R$'s. Number this finite set of requests $L1, \ldots, LN$ in order of increasing timestamp and priority. The lowest priority request $L1$ will be resolved by the DBMP set in finite time since the voting rule prevents it from being deferred by any DBMP. Next consider $L2$. If $L2$ had been deferred at some DBMP $J$, it could have been only because it conflicted with $L1$ or its base variables were too current. The base variables can be too current only if $L2$ was initiated at a site that had already applied $L1$. In either case, $J$ will learn in finite time that $L1$ was resolved. It will then reconsider $L2$ which since it is now the lowest priority request will be resolved in finite time. By similar reasoning it follows that each of the $Li$ will be resolved in finite time. After $LN$ is resolved, $R$ will be the lowest priority request and voting on it can no longer be deferred at any DBMP. Now, any particular case will involve only requests corresponding to some subset of all the timestamps less than $R$'s. The above argument, somewhat simplified since there are fewer requests to consider, remains valid. This establishes Claim 2 for pure daisy chaining.

*Argument* (for daisy chaining with timeout and retransmit). The argument here differs only slightly from the above. The differences result from the way the REJ vote is interpreted by the request resolution rule. Because REJ and PASS votes are equivalent, the argument is similar but slightly shorter than that for pure daisy chaining. As above, the assumed finite time for pairwise DBMP communication ensures that all requests will accumulate sufficient votes for resolution.

A guaranteed finite time for pairwise DBMP communication is necessary because at any given time communication between a given pair of DBMPs may not be possible due to network or host failures. It is possible for failures and recoveries to occur in a way that prevents a request from ever being resolved. For example, a two DBMP system in which the DBMPs are never up at the same time can never accept requests since the two DBMPs never interact. In practice, such failure and recovery patterns are extremely unlikely. Therefore, the finite-time condition for pairwise DBMP communication is a reasonable assumption for a real set of DBMPs.

*Claim* 3. All copies of the data base converge to the same value.

*Argument.* When a DBMP accepts an update it is obligated by the request resolution rule to notify every other DBMP. The update application rule ensures that the value for each data element in a database copy is the value assigned by the accepted update with the most recent timestamp known to the DBMP. The reliable communication mechanism (assumed in Section 2) guarantees that all accepted updates eventually become known to all DBMPs.

Next, we wish to show that the algorithm implements mutual exclusion for accepted updates. To do this we must be more precise about what we mean.

First, consider a DBMP set for which all database copies are initially identical and assume that the set accepts the updates $R1$, $R2$, ..., $RN$. Next, consider a single centralized database which is initially identical to the DBMP database copies. Assume that the transaction sequence:

$$L\ W1\ UL\ L\ W2\ UL \ldots L\ WN\ UL$$

is run against this database, where $L$ and $UL$ are operations which set and clear a lock that controls access to the database and $Wi$ is an update transaction which first reads the base variables $(Bj)$ of some $Rj$, next computes new values for the update variables $(Uj)$ of $Rj$ using the algorithm used by the AP that submitted $Rj$, and then updates the $Uj$. The lock and unlock operations are explicitly shown in the sequence to emphasize that it is run serially, only one transaction at a time.

*Claim* 4. Given a set of updates, $R1$, ..., $RN$, accepted by a DBMP set, it is possible to construct a sequence

$$L\ W1\ UL\ L\ W2\ UL \ldots L\ WN\ UL$$

where each $Wi = Rj$ and $Wi \neq Wk$ for $i \neq k$ such that the values assigned by the $Wi$ in the single copy database system are identical to the values assigned by the corresponding $Rj$ in the DBMP system.

When transaction $Wj$ is performed on the single copy database all previous $Wi$ $(i < j)$ have been performed. In particular, any base variables of $Wj$ assigned values by previous $Wi$'s have been updated since all $Wi$'s have been completely processed. The analogous situation for the DBMP set is not necessarily true. That is, when the base variables at some DBMP $X$ were read to initiate $Rj$, it is not necessarily true that all previous $Ri$ which updated them had been applied to $X$'s database copy.

It follows from Claim 4 that the effect of operating the majority consensus algorithm for a set of updates on the database copies is equivalent to the effect of operating a locking algorithm for the same set of updates on a single copy database. That is, the effect on the database copies of any set of updates accepted by the DBMP set is equivalent to applying the updates serially in a noninterleaved fashion to a single copy database. Since serial application using a locking discipline implements mutual exclusion, it follows that the majority consensus algorithm implements mutual exclusion for accepted updates. Furthermore, since the total set of updates accepted by the majority consensus algorithm is *serializable* [9], if APs initiate only updates that preserve the internal consistency of the database, the DBMP set will operate to preserve that consistency.

*Argument.* An induction argument will be used to show that a sequence of the accepted updates can be generated such that each $Ri$ in the sequence was initiated at a DBMP that had applied (at least the most recent of) all $Rk$ preceding it in the sequence that updated its base variables. More specifically, let $Tj$ be the timestamp generated for $Rj$ and $Tvj$ be the timestamp for base variable $v$ in $Bj$; we wish to generate a sequence with the property that for each $Ri$ in it, $Tvi \geq Tk$ for all $Rk$ that precede $Ri$ for which $v$ is in $Bi$ and in $Uk$, and $Ti > Tvk$ for all $Rk$ which precede $Ri$ for which $v$ is in $Ui$ and in $Bk$. Such a sequence of all the accepted updates satisfies the claim since any update $Rk$ that assigned values to the base variables that are read by another update $Ri$ precedes $Ri$ in the sequence, and any $Ri$ that will assign, but has not yet assigned, values to the base variables that are read by another update $Rk$ follows $Rk$ in the sequence. Therefore, the values assigned by any $Ri$ are identical to those it would assign if all updates that precede it in the sequence were run in a noninterleaved fashion against a single copy database.

The procedure for generating the sequence will be to start with $R1$ to make a sequence of length 1 with the property; and then to iteratively consider each $Rn$ and order it with respect to the previously ordered $R1, \ldots, Rn - 1$ to generate a sequence of length $n$ with the desired property. Generation of a sequence that satisfies the claim will be complete when the final request is added to the sequence.

Before proceeding we need to introduce the notions of acceptance set and time of acceptance and to establish a result concerning the relation between the time of acceptance for requests and the positions they may occupy relative to one another in a sequence with the desired property.

We define the *resolution set* for a request to be the set of DBMPs whose votes contributed to the resolution of the request. The *resolving DBMP* for a resolution set is the DBMP that accepted or rejected the request. The *time of resolution* for a resolution set is the time at which the resolving DBMP resolved the request;

$$Tres = (\max(\text{time}, C(T) + 1), r)$$

where "time" is the time obtained by the resolving DBMP from its local clock, $C(T)$ is the $c$-part of the request timestamp, and $r$ is the resolving DBMP. Notice that for daisy chaining with timeout and retransmit a request may have more than one resolution set.

We define the *time of acceptance* for an accepted request to be its minimum time of resolution:

$$Ta = \min(\{Tres\}),$$

the *acceptance set* for a request to be the resolution set corresponding to its time of acceptance, and the *accepting DBMP* to be the resolving DBMP for its acceptance set. Even with a timeout and retransmit discipline an accepted request has a single acceptance set and time of acceptance.

Next we define a constraint concerning where conflicting updates may appear relative to one another in a sequence. *Ri must precede Rj with respect to v* if either:

*P*1: $v$ is in $Ui$ and in $Bj$ and $Tvj \geq Ti$ (i.e. $Ri$ modified a base variable of $Rj$ before $Rj$ read it); or

*P*2: $v$ is in $Bi$ and in $Uj$ and $Tvi < Tj$ (i.e. $Rj$ modified a base variable of $Ri$ after $Ri$ read it).

*Result.* If $Ri$ must precede $Rj$ with respect to some $v$ then $Ri$ was accepted before $Rj$.

*Argument for Result.* By the request resolution rule there must be a DBMP $X$ in the acceptance set of $Ri$ and $Rj$ that voted OK for both requests.

Suppose $X$ voted first on $Ri$. In this case the voting rule prevents $X$ from voting OK on $Rj$ until $Ri$ is resolved (since $Ri$ and $Rj$ conflict). Therefore, $Ri$ was accepted before $X$ voted on $Rj$ and hence before $Rj$ was accepted.

Suppose $X$ voted first on $Rj$. In the following, let $Tvdbi$ be the timestamp for $v$ in $X$'s copy of the database when $X$ votes for $Ri$. As above the voting rules prevent $X$ from voting OK on $Ri$ until $Rj$ is accepted and $Rj$'s results are in $X$'s database copy. Since $Ri$ must precede $Rj$ with respect to $v$, either *P*1 or *P*2 must be the case.

Suppose *P*1 ($Tvj \geq Ti$). $Tvdbi \geq Tvj$ since, when $X$ voted OK on $Rj$, $Tvdbj = Tvj$ ($v$ is in $Bj$) and the update application rule prevents the timestamp for $v$ in $X$'s database from decreasing. Since $Tvj \geq Ti$, it follows that $Tvdbi \geq Ti$. By the timestamp generation rule $Ti > Tvi$. Therefore, $Tvdbi > Tvi$ ($v$ is in $Ui$ and hence $Bi$) which requires $X$ to vote REJ on $Ri$. Hence if $X$ voted first on $Rj$, *P*1 could not be the case.

Suppose *P*2 ($Tvi < Tj$). Since $Rj$ must have been applied when $X$ votes on $Ri$, $Tvdbi \geq Tj$ ($v$ is in $Uj$). Since $Tj > Tvi$, it follows that $Tvdbi > Tvi$. This requires $X$ to vote REJ on $Ri$. Hence if $X$ voted first on $Rj$, *P*2 could not be the case.

Therefore, $X$ could not have voted first on $Rj$. This establishes the result. Now we can proceed with the construction of a sequence for Claim 4.

$i = 1$. Select $R1$ to form a sequence of length 1. This sequence has the desired property since it contains only a single element.

*Induction Step.* Assume any $n - 1$ of the requests, $R1, \ldots, Rn - 1$, can be arranged in a sequence $Sn - 1$:

$$Sn - 1\colon \quad Rp \ldots Rq$$

with the desired property. The induction step is to show that $Rn$ can be added to the requests in $Sn - 1$ to form a new sequence $Sn$ with the desired property. The approach is to identify a request, $Rx$ (either $Rn$ or one of the requests in $Sn - 1$), which must (or may) be at the end of the new sequence $Sn$, and then to use the induction assumption to assert that the remaining requests can be arranged in a sequence of length $n - 1$.

*Step* 1. Identify the $Ri$ nearest the end of $Sn - 1$ for which there is a $v$ in $Bn$ and in $Ui$ and $Tvn < Ti$ or a $v$ in $Un$ and in $Bi$ and $Tvi \geq Tn$. If there is no such $Ri$, then the new sequence is $Sn = Sn - 1\ Rn$. If there is, set $Rx$ to be $Ri$. By the result above, $Rn$ was accepted before $Rx$ since $Rn$ must precede $Rx$ with respect to $v$.

*Step* 2. Identify the $Ri$ nearest the end of $Sn - 1$ for which there is a $v$ such that $Rx$ must precede $Ri$ with respect to $v$. If there is no such $Ri$, begin Step 3. If there is, by the result above $Rx$ was accepted before $Ri$. Set $Rx$ to this $Ri$. $Rn$ was accepted before this new $Rx$, since it was accepted before the old $Rx$ which was accepted before the new $Rx$. Repeat this step.

*Step* 3. There is no $Ri$ following $Rx$ in $Sn - 1$ for which there is a $v$ such that $Rx$ must precede $Ri$ with respect to $v$. Since $Rx$ updates no variables that are base variables of any $Ri$ following it in $Sn - 1$ and no $Ri$ following $Rx$ in $Sn - 1$ updates any base variable of $Rx$, $Rx$ can be moved to the end of $Sn - 1$:

$$Sn - 1\text{:}\quad Rp \ldots Rq\ Rx.$$

*Step* 4. Form $S = Sn - 1\ Rn$:

$$S\text{:}\quad Rp \ldots Rq\ Rx\ Rn.$$

We know from the way $Rx$ was found that $Rn$ was accepted before $Rx$. If we can show that $Rx$ and $Rn$ may (or must) be interchanged, we will establish Claim 4 since we will have

$$S\text{:}\quad Rp \ldots Rq\ Rn\ Rx$$

and by the induction assumption any collection of $n - 1$ updates, and in particular $Rp \ldots Rq\ Rn$, can be arranged in a sequence with the desired property. The new sequence we are trying to generate, $Sn$, is this rearranged sequence of length $n - 1$ followed by request $Rx$.

There are three cases to consider: (a) there is a $v$ in $Un$ and $Bx$; (b) there is a $v$ in $Bn$ and $Ux$; (c) neither (a) nor (b).

If (c) is the case then $Rn$ and $Rx$ may be interchanged in $Sn$ since they do not conflict.

Suppose (a). Consider the site $Y$ in the acceptance sets of $Rn$ and $Rx$ which voted OK on both requests. If $Y$ voted first on $Rn$, then $Rn$'s results are in $Y$'s database when it votes on $Rx$. To vote OK on $Rx$, $Tvx$ must be $\geq Tn$ for all $v$ in $Un$ and $Bx$. Therefore, $Rn$ must precede $Rx$ in the sequence and should be interchanged with it. Now suppose $Y$ voted first on $Rx$. $Rx$'s results must have been in $Y$'s database when it voted on $Rn$. This could only occur if $Rx$ was accepted before $Rn$ which was not the case. Therefore if (a), $Rn$ and $Rx$ must be interchanged.

Suppose (b). Consider $Y$. From above we know that $Y$ voted on $Rn$ first. When $Y$ voted on $Rn$, $Tvn = Tvdbn$. When it later voted on $Rx$, $Tvx = Tvdbx \geq Tvdbn = Tvn$ since the update application rule ensures that $Tvdb$ can never decrease. (Since $v$ is in $Ux$, it is also in $Bx$.) By the timestamp generation rule $Tx > Tvx$

from which it follows that $Tx > Tvn$. Therefore, $Rx$ must follow $Rn$ in the sequence and should be interchanged with it.

*Claim* 5. The timestamp generation rule prevents the occurrence of sequencing anomalies.

*Argument.* Let $A$ and $B$ be two update requests. Assume that first $A$ is requested and accepted by the DBMP set and then $B$ is requested and accepted. To show that sequencing anomalies cannot occur we must show that the value of any database element which is an update variable of $A$ and $B$ will be specified by $B$. That is, we wish to show for the timestamps, $Ta$ and $Tb$, generated for $A$ and $B$:

$$Tb > Ta.$$

Recall that the update variables for a request are a subset of the base variables. Therefore, since $A$ and $B$ have update variables in common, there must be a variable $v$ that is a base variable of $B$ and an update variable of $A$. Since $B$ is initiated after $A$ was accepted, $B$ must have been initiated at a DBMP $X$ that had already applied $A$; otherwise, the timestamp of $v$ in request $B$ would be obsolete, leading to $B$'s eventual rejection. When $B$ was initiated at $X$ the timestamp of $v$ in $X$'s copy of the database must have been at least $Ta$. The timestamp generation rule guarantees that $Tb$ is at least $Ta + 1$. Therefore, $Tb > Ta$.

Claim 5 means that it is possible for a DBMP set to properly sequence conflicting update events without requiring that the local DBMP clocks used in the generation of update timestamps be synchronized. A local DBMP clock can run at a different rate than other DBMP clocks; it can even run at a variable rate, or not run at all. Claim 5 is an important result because it is difficult to synchronize clocks in a distributed environment.

We note that it does not follow from this result that any two events initiated at different DBMPs in a system with asynchronous DBMP clocks can be properly sequenced. It only ensures that events with something in common (i.e. those that conflict with one another) can be sequenced. The ability of the algorithm to properly sequence updates that modify the same variable or sets of variables is a consequence of the requirement that the update variables be a subset of the base variables. In intuitive terms, the variables in common between conflicting updates are the handles which enable the voting DBMPs to properly sequence seemingly asynchronous events. The reader interested in more on the subject of event ordering and clock synchronization is referred to [10].

## 5. COST OF THE ALGORITHM

It is possible to identify the following costs which are incurred as a result of using the majority consensus algorithm:

(1) *Communication.* A number of interprocess messages must be exchanged to accomplish an update.

(2) *Computation.* Values for the update variables must be computed. The race resolution mechanism occasionally requires that an update request be rejected. If the requesting AP wishes to accomplish an update that has been

rejected, the AP must, in general, first recompute it and then resubmit it as another request.

(3) *Delay.* It takes some time for the DBMP set to resolve an update request.

(4) *Storage.* Timestamps must be stored in the database with the data elements. In addition, while a request is being resolved, DBMPs where it is pending must maintain a record of it until it is resolved and they can safely discard it.

This section examines the communication and computation costs imposed by the algorithm. A pure daisy chaining communication discipline is assumed.

Consider an $n$ DBMP system. The number of messages required to accomplish an update under best case conditions (i.e. no conflicts with other update requests, no DBMP failures) is:

| | |
|---|---|
| $3$ | to initiate the update (AP $\langle - \rangle$ DBMP messages to request variables, transmit variables, request update) |
| $+ \lceil n/2 \rceil$ | to achieve a consensus[11] (inter-DBMP messages); |
| $+ \; n - 1$ | to notify the DBMP set of acceptance (inter-DBMP messages); |
| $+ \; 1$ | to notify AP of acceptance (DBMP $- \rangle$ AP message) |

or $n + \lceil n/2 \rceil + 3$ messages.

If there are conflicts, the votes of more than $\lceil n/2 \rceil$ DBMPs may be required to resolve a request. Each additional DBMP vote requires an additional message. Given that an update is accepted the first time it is requested, in the worst case every DBMP would have to vote before the request could be accepted. This would require $n - 1$ inter-DBMP messages. Therefore, in this case, a request would require $2n + 2$ messages to be accepted.

If a timeout and retransmit discipline is being used and if a DBMP that has voted on a request fails before it can forward the request, additional messages may be generated by the request timeout mechanism.

The best case figure of $n + \lceil n/2 \rceil + 3$ compares favorably with other techniques one might consider for managing distributed, redundant databases.

An update algorithm is described in [8] which guarantees mutual consistency but cannot ensure internal consistency of database copies. The number of messages required by that mechanism to accomplish an update is:

| | |
|---|---|
| $3$ | for an AP to initiate an update; |
| $+ \; 1$ | for the initiating DBMP to acknowledge the update; |
| $+ \; n - 1$ | to communicate the update to the other DBMPs |

or $n + 3$ messages. The difference of $\lceil n/2 \rceil$ is exactly the number of messages required to reach a majority consensus and can be regarded as the cost of insuring internal consistency.

It is interesting to note that update algorithms which use centralized control

[11] $\lceil x \rceil$ is the "ceiling" function, the least integer $\geq x$.

also require $n + 3$ interprocess messages. To see this assume that the central control point resides in one of the DBMPs. As in the distributed control algorithm, an AP and the central DBMP must exchange three messages to initiate the update; $n - 1$ messages are required to distribute the update to the other DMBPs; and one message is required to inform the AP that the update has occurred.

It is possible to imagine algorithms that involve locking each copy of the database for the duration of the activity required to process an update. We consider such a mechanism only for purposes of comparison: it is clearly less robust with respect to component failures and outages than the majority consensus mechanism; furthermore, it may be very difficult to specify such a locking algorithm that is deadlock-free. The number of messages required by a locking algorithm to accomplish an update would be:

| | |
|---|---|
| $n$ | to lock each copy of the database; |
| $+ 1$ | to obtain the base variables; |
| $+ n$ | to perform the update and unlock the database copies |

or $2n + 1$ messages. Thus even in the worst case $(2n + 2)$ the majority consensus algorithm compares well with a simple lock-update-unlock scheme.

The cost of accomplishing an update includes both computation and communication costs. Let $C$ be the cost of computing an update and $M$ be the cost of transmitting a single message; for simplicity, assume that all messages cost the same.

Using the results from above, the cost, $C0$, of an update that is accomplished without rejection is

$$C + (n + \lceil n/2 \rceil + 3)M \leq C0 \leq C + 2(n + 1)M.$$

If we define

$$C0 \text{ min} = C + (n + \lceil n/2 \rceil + 3)M$$

$$C0 \text{ max} = C + 2(n + 1)M$$

then the bounds on the cost, $C1$, of an update that is accomplished with a single rejection and resubmission can be shown to be:

$$C0 \text{ min} + C + 2M \leq C1 \leq C0 \text{ max} + C + 2nM.$$

Intuitively, these bounds can be explained as follows. In the best case, the first update request will be rejected by the initiating DBMP; the $2M$ accounts for the messages from the DBMP to the AP to reject the request and the message from the AP to the DBMP to resubmit the update;[12] $C$ represents the cost of recomputing the update. In the worst case, all DBMPs must vote before the first update request is rejected, requiring $n - 1$ inter-DBMP messages and an additional $n - 1$ inter-DBMP messages by the rejecting DBMP to communicate the rejection to the other DBMPs.

In general, it can be shown that the cost, $Ck$, of an update that is rejected and

[12] This assumes the message to notify the AP that the request has been rejected includes the current values and timestamps for the base variables; this enables the AP to resubmit the update without re-requesting the base variables. If the rejection is to prevent a possible deadlock, the values and timestamps returned may not be current.

resubmitted to the DBMP set $k$ times before it is accomplished is:

$$C0\ \min + k(C + 2M) \le Ck \le C0\ \max + k(C + 2nM).$$

## 6. THE PROBLEM OF MEMORY LOSS

Correct operation of the majority consensus algorithm requires that information regarding the state of the database system is never lost by any DBMP. We assume that anything worth remembering by a DBMP, such as the database itself and unresolved update requests, is maintained by the DBMP on a nonvolatile storage medium, such as disk, which normally survives host system failures. We further assume that the DBMP can determine when data being moved from volatile (e.g. core) to nonvolatile storage has been completely copied to the nonvolatile medium.

A DBMP is said to have "lost memory" if it has forgotten updates which have been accepted or if it has forgotten how it has voted on currently unresolved update requests. A DBMP memory loss would occur if the information on the nonvolatile storage medium used by the DBMP is destroyed.

If a DBMP that has lost memory is permitted to vote on update requests, that DBMP could cause the majority consensus algorithm to malfunction. This could happen if: (1) the DBMP votes OK for a request which conflicts with accepted updates it has forgotten, thereby possibly enabling the request, which should be rejected, to achieve a majority consensus; or (2) when asked to vote on an unresolved request it has previously voted on and forgotten, the DBMP votes differently (e.g. votes OK rather than PASS), thereby possibly causing the request to be both accepted and rejected.

By itself, a DBMP has no way of determining whether it has lost memory. We assume that memory loss occurs as the result of some catastrophic event at the database site and that in such a case the information critical to DBMP operation is restored by a human operator from a backup copy which is possibly out of date. The backup copy would typically be archived on magnetic tape. We assume that whenever the information is backed up in this way, the DBMP is restarted and signaled that a memory loss has occurred. In addition, we assume the DBMP can determine the point of memory loss. That is, we assume that the DBMP keeps a record of timestamps for recent significant events, such as the last update accepted at each other DBMP, on the nonvolatile storage medium and that this record is archived along with the database and also restored after a memory loss occurs.

When a DBMP restarts after a memory loss, it must follow a memory recovery procedure before it can safely vote on update requests. In order to become a voting member of the DBMP set, a DBMP that has lost memory must: (1) recover all updates which the set of DBMPs has accepted since the point of its memory loss (and which have not been forgotten by the entire set of DBMPs); and (2) recover all unresolved update requests which it has voted on since the point of its memory loss.

It can be shown that, in general, a DBMP with memory loss must interact with every other DBMP in order to guarantee recovery of all the information it has lost. Furthermore, it can be shown that a recovery scheme which involves only a simple interaction with each other DBMP, in which such information is requested and transmitted, is insufficient to recover all the lost information.[13]

Below we present a two-pass memory recovery procedure which involves only pairwise interactions among DBMPs. We assert that this memory recovery procedure works correctly when one, several, or all DBMPs have lost memory. However, it is beyond the scope of this paper to prove its correctness.

Let $M$ be the DBMP with memory loss. On the first pass $M$ informs each other DBMP that it is trying to recover from a memory loss. When a DBMP is so informed, it must acknowledge, and in addition, temporarily stop forwarding to other DBMPs unresolved requests that have been voted on by $M$.[14]

On the second pass $M$ requests from each other DBMP, in turn, information concerning updates accepted since the point of $M$'s memory loss and unresolved update requests voted on by $M$. After it supplies $M$ such information, a DBMP may resume forwarding unresolved requests that $M$ has voted on.

If on the second pass $M$ encounters a DBMP that is unaware that $M$ is engaged in the memory recovery procedure, that DBMP has also lost memory (since $M$'s first pass). Should $M$ encounter such a DBMP, it must abort the second pass of the procedure. In such a case, to proceed with its memory recovery, $M$ must repeat the first pass of the procedure, after which it may restart the second pass. When $M$ successfully completes the second pass, it can participate as a voting member of the DBMP set.

## 7. CONCLUDING REMARKS

This paper has presented a "majority consensus" algorithm which represents a new solution to the update synchronization problem for multiple copy databases. Because the responsibility for performing an update is distributed among the collection of processes that manage database copies rather than centralized in a single process, the algorithm can function effectively (i.e. process updates) in the presence of communication and database site outages.

Analysis of the communication and computation costs incurred by the majority consensus algorithm to accomplish an update (when it is unnecessary to reject and resubmit it) shows these costs are not significantly greater than for other more centralized approaches. When the pattern of update activity is such that conflicting update requests occur, these costs increase because more votes are required to resolve requests and because rejected update requests must be resubmitted.

In addition to communication and computation costs, the algorithm imposes a significant short-term storage requirement upon the database sites since each site must remember the state of a pending update request until the request is resolved. The short-term storage required for any application will depend upon the ex-

[13] While one DBMP is attempting to recover memory, it is possible for the other DBMPs to experience memory loss and engage in memory recovery in pathological patterns which would enable unresolved update requests voted on by the original DBMP to remain active in the DBMP set but unrecoverable by any simple one-pass procedure.

[14] This temporary freezing of database activity with respect to these unresolved requests prevents the pathological behavior mentioned in footnote 13.

pected patterns of update activity. In practice, the dominant cost associated with the use of the algorithm is likely to be that incurred to satisfy this short-term memory requirement.

A multiple copy database is one particular type of distributed database. Another type is one which consists of distributed, nonoverlapping segments; that is, a database which is a collection of smaller database segments each of which is singly maintained at a (possibly) different site.[15] Although the data are not redundantly stored for this type of distributed database, in some applications it may be desirable to maintain multiple copies of the catalogs for such a segmented database. For these applications the majority consensus algorithm could be used to handle updates to the database catalog.

A number of interesting questions regarding the use of multiple copy databases, in general, and the use of the majority consensus algorithm, in particular, remain to be answered. These questions include:

(1) How should application processes be programmed to deal with the fact that data found in any given database copy may not be the most current? In some cases it may not be critical that the data are not current. If it is critical, how can a process locate the most current data?
(2) How will the algorithm perform under various patterns of update activity and various patterns of communication system and site outages? For example, given particular activity and outage patterns, what is the probability that an update will be accepted the first time it is submitted; what is the expected number of DBMPs that must vote for an update request to be resolved?
(3) In practice, use of the memory recovery procedure sketched in Section 6 could be expensive in terms of the storage required to maintain update history information at each DBMP site. What strategies can be used to minimize the extent of the history information that is maintained at each site? The memory recovery procedure that was presented is interesting in that, like the majority consensus algorithm, it can be made extremely robust because it incorporates distributed control. However, since memory loss by a DBMP is likely to be a rare occurrence (relative to communication system and site failures), a simpler, centralized recovery procedure may be adequate in most situations.

## APPENDIX

This Appendix includes a number of examples chosen to illustrate various aspects of the update algorithm. Before presenting them it is necessary to specify in some detail the messages exchanged among APs and DBMPs. The following messages are used in the examples:

DBMP ⟨–⟩ AP messages:
RV — Request Variable values and timestamps (AP to DBMP).
VAR — VARiables and timestamps (DBMP to AP).

[15] These two types represent extremes. Some applications may call for "intermediate" types; for example, a database comprised of a collection of smaller segments some, but not all, of which are redundantly maintained.

| | |
|---|---|
| RU | — Request Update (AP to DBMP). |
| UA | — Update Accepted (DBMP to AP). |
| UR | — Update Rejected (DBMP to AP). |

Inter-DBMP messages:

| | |
|---|---|
| RC | — Request Consensus on specified update request. |
| DO | — The specified update request has been accepted; enter it into your copy of the database. |
| REJ | — The specified update request has been REJected. |

For each of the examples that follow a number of different sequences of events are possible; only one sequence is presented for each example. The following notation is used in the examples:

- *X*−⟩*Y*:*Z* represents transmission of message *Z* to process *Y* by process *X*.
- [*A* / *B* / *C*] indicates the event sequence in which event *A* is followed by event *B* which is followed by event *C*.
- [*A* & *B*] indicates that events *A* and *B* occur concurrently.
- The update request status "——" indicates that the update request is currently unknown at the DBMP in question. The status "XX" indicates that the DBMP in question is down.
- ok@12 (pa@12, rj@12) means that DBMPs 1 and 2 have voted OK (PASS, REJ) on the request. Similarly do@2 (rej@2) means that DBMP 2 accepted (rejected) the update request.
- DONE means that the DBMP has performed the update.
- REJD means that the DBMP considers the request as rejected.
- "*" indicates that the DBMP is actively trying to forward information regarding the request; *ok means that it is trying to forward an RC message; *DONE means that it is trying to complete sending DO messages; *REJD means that it is trying to complete sending REJ messages.

The first three examples assume pure daisy chain communication. The fourth example uses daisy chaining with timeout and retransmit.

*Example* 1: Normal Update with no Conflict. Consider three DBMPs which manage a database which includes a variable $x$. Assume that an AP wishes to do the update:

$$x := x + 1.$$

Further, suppose that $x$ is current in all copies of the database, and that its value is 3. Let the update requested be called $A$. $A$ has a single base variable, $x$, and a single update variable, $x$. If accepted, $A$ will change the value of $x$ to 4.

The sequence of events that occurs and how the status of the request $A$ as seen by each DBMP evolves as the DBMPs work to accomplish the update is shown below:

| | DBMP 1 | DBMP 2 | DBMP 3 |
|---|---|---|---|
| Status of: | | | |
| *A* | —— | —— | —— |

[AP−⟩1:RV(*x*) / 1−⟩AP:VAR(*x*) / AP−⟩1:RU(*A*) / 1 votes OK]

| | | | |
|---|---|---|---|
| *A* | *ok@1 | —— | —— |

[1−⟩2:RC(*A*) / 2 votes OK]

| | | | |
|---|---|---|---|
| *A* | ok@1 | *ok@12 | —— |

[2 accepts *A* / 2−⟩1:DO(*A*) / 2−⟩AP:UA(*A*)]

| | | | |
|---|---|---|---|
| *A* | DONE | *DONE | —— |
| | do@2 | do@2 | |

[2−⟩3:DO(*A*)]

| | | | |
|---|---|---|---|
| *A* | DONE | DONE | DONE |
| | do@2 | do@2 | do@2 |

[1, 2, 3 discard request *A*]

*Example* 2: Concurrent Conflicting Updates. This is the example from Section 1. There are three DBMPs which manage a database that includes variables $x$, $y$, and $z$. Assume all databases are current and $x = y = z = 1$ in all copies of the database. Assume that AP1 initiates update $A$ and that AP2 initiates update $B$:

$$\begin{aligned} &A\colon x := -1, \quad && y := 3 \\ &B\colon y := -1, \quad && z := 3. \end{aligned}$$

The base variables of $A$ are $x$, $y$, $z$ and the update variables are $x$, $y$; $B$'s base variables are $x$, $y$, $z$ also, and its update variables are $y$, $z$. $A$ and $B$ conflict.

In the following, $A$ is accepted causing $B$ to be rejected. AP2 then chooses to reinitiate its update (called $B'$ to distinguish it from the AP2's original request) which is then accepted. We assume that the priority of request $A$ is greater than that of $B$.

| Status of: | DBMP 1 | DBMP 2 | DBMP 3 |
|---|---|---|---|
| *A* | —— | —— | —— |
| *B* | —— | —— | —— |

[AP1−⟩1:RV(*xyz*) & AP2−⟩3:RV(*xyz*) / 1−⟩AP1:VAR(*xyz*) & 3−⟩AP2:VAR(*xyz* /AP1−⟩1:RU(*A*) & AP2−⟩3:RU(*B*) / 1 votes *A*-OK & 3 votes *B*-OK]

| | | | |
|---|---|---|---|
| *A* | *ok@1 | —— | —— |
| *B* | —— | —— | *ok@3 |

[1−⟩2:RC(*A*) & 3−⟩1:RC(*B*) / 2 votes *A*-OK & 1 votes *B*-PASS]

| | | | |
|---|---|---|---|
| *A* | ok@1 | *ok@12 | —— |
| *B* | *ok@3pa@1 | —— | ok@3 |

[2 accepts *A* & 1−⟩2:RC(*B*) / 2−⟩AP1:UA(*A*) & 2−⟩1:DO(*A*) & 2 rejects *B*]

| | | | |
|---|---|---|---|
| A | DONE | *DONE | —— |
| | do@2 | do@2 | —— |
| *B* | ok@3pa@1 | *REJD | ok@3 |
| | | rej@2 | |

[2−⟩3:DO(*A*) & 2−⟩1, 3:REJ(*B*) & 2−⟩AP2:UR(*B*)]

| | | | |
|---|---|---|---|
| *A* | DONE | DONE | DONE |
| | do@2 | do@2 | do@2 |
| *B* | REJD | REJD | REJD |
| | rej@2 | rej@2 | rej@2 |

[1, 2, 3 discard *A* and *B*]

| | | | |
|---|---|---|---|
| *A* | —— | —— | —— |
| *B* | —— | —— | —— |

[AP2–⟩2:RV(*xyz*) / 2–⟩AP2:VAR(*xyz*) / AP2–⟩2:RU(*B*′) / 2 votes *B*′-OK]

| | | | |
|---|---|---|---|
| *B*′ | —— | *ok@2 | —— |

[2–⟩3:RC(*B*′) / 3 votes *B*′-OK / 3 accepts *B*′ / . . . etc.]

*Example* 3: Deadlock Avoidance. Assume three DBMPs which manage a database which includes the variables $x, y$, and $z$. Assume that all copies of the database are current and that $x = 1$, $y = 2$, and $z = 3$. Assume that three application programs attempt the updates:

$$\begin{aligned} &A\colon x := y^*z \quad \text{(by AP1)} \\ &B\colon y := z + x \text{ (by AP2)} \\ &C\colon z := x - y \text{ (by AP3).} \end{aligned}$$

Update $A$ would change $x$ to 6; $B$ would change $y$ to 4; $C$ would change $z$ to $-1$. The base variables of all three requests are $x, y, z$; the update variables are such that each request conflicts with each of the others. In the following scenario the DBMPs act first to reject $C$ in order to prevent a possible deadlock, next to accept $B$, and finally, to reject $A$ because it conflicts with $B$.

| Status of: | DBMP 1 | DBMP 2 | DBMP 3 |
|---|---|---|---|
| *A* | —— | —— | —— |
| *B* | —— | —— | —— |
| *C* | —— | —— | —— |

[. . . AP1–⟩1:RU(*A*) & AP2–⟩2:RU(*B*) & AP3–⟩3:RU(*C*) / 1, 2, 3 vote OK on *A*, *B*, *C*]

| | | | |
|---|---|---|---|
| *A* | *ok@1 | —— | —— |
| *B* | —— | *ok@2 | —— |
| *C* | —— | —— | *ok@3 |

[1–⟩2:RC(*A*) & 2–⟩3:RC(*B*) & 3–⟩1:RC(*C*) / 2 defers *A* & 3 defers *B* & 1 votes *C*-PASS]

| | | | |
|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | —— |
| *B* | —— | ok@2 | DEFR, ok@2 |
| *C* | *ok@3pa@1 | —— | ok@3 |

[1–⟩2:RC(*C*) / 2 votes *C*-PASS / 2 rejects *C*]

| | | | |
|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | —— |
| *B* | —— | ok@2 | DEFR, ok@2 |
| *C* | ok@3pa@1 | *REJD rej@2 | ok@3 |

[2–⟩1, 3:REJ(*C*) & 2–⟩AP3:UR(*C*) / 3 votes *B*-OK / 3 accepts *B*]

| | | | |
|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | —— |
| *B* | —— | ok@2 | *DONE do@3 |
| *C* | REJD rej@2 | REJD rej@2 | REJD rej@2 |

[3–⟩1, 2:DO(*B*) & 3–⟩AP2:UA(*B*) / 1, 2, 3 discard *C* & 1, 2, reject *A*]

| | | | |
|---|---|---|---|
| *A* | *REJD rej@1 | *REJD rej@2 | —— |
| *B* | DONE | DONE | DONE |

| | | | |
|---|---|---|---|
| | do@3 | do@3 | do@3 |
| *C* | — | — | — |

[1, 2, 3 discard *B* / 1–⟩2, 3:REJ(*A*) & 1–⟩AP1:UR(*A*) & 2–⟩1, 3:REJ(*A*) & 2–⟩AP1:UR(*A*, *xyz*)]

| | | | |
|---|---|---|---|
| *A* | REJD | REJD | REJD |
| | rej@12 | rej@12 | rej@12 |
| *B* | — | — | — |
| *C* | — | — | — |

[1, 2, 3 discard *A*]

*Example* 4: Updating in the Presence of DBMP Crashes. For this example assume a five DBMP system and that all database copies are current. Further assume that DBMPs 4 and 5 are initially down and that when DBMPs crash and later come up they do so without loss of memory. Suppose that conflicting updates *A* and *B* are initiated at DBMPs 1 and 3, respectively. The following illustrates a scenario in which various DBMPs crash and return as the set of DBMPs act to accept *B* and reject *A*.

| Status of: | DBMP 1 | DBMP 2 | DBMP 3 | DBMP 4 | DBMP 5 |
|---|---|---|---|---|---|
| *A* | — | — | — | XX | XX |
| *B* | — | — | — | XX | XX |

[AP1–⟩1:RU(*A*) & AP2–⟩3:RU(*B*) / 1, 3 vote OK on *A*, *B*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | *ok@1 | — | — | XX | XX |
| *B* | — | — | *ok@3 | XX | XX |

[3–⟩1:RC(*B*) / 1 votes *B*-PASS / 1–⟩2:RC(*B*) / 2 votes *B*-OK / 1–⟩2:RC(*A*) / 2 defers *A*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | — | XX | XX |
| *B* | ok@3pa@1 | *ok@23pa@1 | ok@3 | XX | XX |

[2 crashes / 1 times out *A*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | *ok@1 | XX | — | XX | XX |
| *B* | ok@3pa@1 | XX | ok@3 | XX | XX |

[1–⟩3:RC(*A*) / 3 defers *A* / 4, 5 up / 3 times out *B*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | XX | DEFR, ok@1 | — | — |
| *B* | ok@3pa@1 | XX | *ok@3 | — | — |

[3–⟩4:RC(*B*) / 4 votes *B*-OK]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | XX | DEFR, ok@1 | — | — |
| *B* | ok@3pa@1 | XX | ok@3 | *ok@34 | — |

[3, 4 crash / 1 times out *A*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | *ok@1 | XX | XX | XX | — |
| *B* | ok@3pa@1 | XX | XX | XX | — |

[1–⟩5:RC(*A*) / 5 votes *A*-OK / 2, 3, 4 up]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | DEFR, ok@1 | — | *ok@15 |
| *B* | ok@3pa@1 | *ok@23pa@1 | ok@3 | *ok@34 | — |

[Note that *B* has been resolved but that no single DBMP is aware of that yet.
5–⟩4:RC(*A*) & 4–⟩5:RC(*B*) / 4 defers *A* & 5 votes *B*-PASS]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | DEFR, ok@1 | DEFR, ok@1 | DEFR, ok@15 | ok@15 |
| *B* | ok@3pa@1 | *ok@23pa@1 | ok@3 | ok@34 | *ok@34pa@5 |

[2–⟩5:RC(*B*) / 5 accepts *B* / 5–⟩1, 2, 3, 4:DO(*B*) & 5–⟩AP2:UA(*B*) / 2, 3, 4 vote *A*-REJ]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | *rj@2ok@1 | *rj@3ok@1 | *rj@4ok@15 | ok@15 |
| *B* | DONE | DONE | DONE | DONE | DONE |
| | do@5 | do@5 | do@5 | do@5 | do@5 |

[2−⟩3:RC(*A*) & 3−⟩4:RC(*A*) & 4−⟩3:RC(*A*) / 3 notices rj@234 for *A* and rejects *A*]

| | | | | | |
|---|---|---|---|---|---|
| *A* | ok@1 | rj@2ok@1 | *REJD | rj@34ok@15 | ok@15 |
| | | | rej@3 | | |
| *B* | DONE | DONE | DONE | DONE | DONE |
| | do@5 | do@5 | do@5 | do@5 | do@5 |

[3−⟩1, 2, 4, 5:REJ(*A*) / 1, 2, 3, 4, 5 discard *A*, *B*]

## ACKNOWLEDGMENTS

Paul Johnson and Harry Forsdick contributed to the formulation of the ideas in this paper. In addition, conversations with Rick Schantz, Vera Koetelboeter, Ray Tomlinson, and Maurice Maybury were helpful.

## REFERENCES

1. Shapiro, R.M., and Millstein, R.E. The NSW reliability plan. Rep. CA-7701-1411, Massachusetts Computer Associates, June 1977.
2. Rothnie, J.B., Goodman, N., and Bernstein, P.A. The redundant update methodology of SDD-1: A system for distributed data bases (the fully redundant case). Tech. Rep. CCA-77-02, Computer Corp. of America, Cambridge, Mass., June 1977.
3. Alsberg, P.A., and Day, J.D. A principle for resilient sharing of distributed resources. Rep. from Ctr. for Advanced Comput., U. of Illinois at Urbana-Champaign, Urbana, Ill., 1976.
4. Henderson, Jr., D.A., and Myer, T.H. Issues in message technology. Proc. Fifth Data Communication Symp., Snowbird, Utah, Sept. 1977, pp. 6-1-6-9.
5. Roberts, L.G., and Wessler, B.D. Computer network development to achieve resource sharing. Proc. AFIPS 1970 SJCC, AFIPS Press, Montvale, N.J., pp. 543–549.
6. Metcalfe, R., and Boggs, D. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), 395–404.
7. Cerf, V., and Kahn, R. A protocol for packet network interconnection. *IEEE Trans. Comm. Comm-22*, 5 (May 1974), 637–648.
8. Johnson, P., and Thomas, R. The maintenance of duplicate data bases. Network Information Center (NIC) Document #31507, ARPA Network Working Group Request for Comments (RFC) #677, Jan. 1975.
9. Eswaren, K.P., Gray, J.N., Lorie, B.A., and Traigen, I.L. The notions of consistency and predicate locks in a database system. *Comm. ACM* 19, 11 (Nov. 1976), 624–633.
10. Lamport, L. Time, clocks and the ordering of events in a distributed system. Rep. CA-7603-2911, Massachusetts Computer Associates, March 1976; also submitted to *Comm. ACM.*

Received January 1978; revised May 1978