BBN Report No. 3340

July 1976

A SOLUTION TO THE UPDATE PROBLEM FOR MULTIPLE COPY DATA BASES WHICH USED DISTRIBUTED CONTROL

Robert H. Thomas

This work was supported by the Defense Advanced Research Projects Agency of the Department of Defense. It was monitored by the Office of Naval Research under Contract No. N00014-75-C-0773.

A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control

Robert H. Thomas
Bolt Beranek and Newman, Inc.*

ABSTRACT

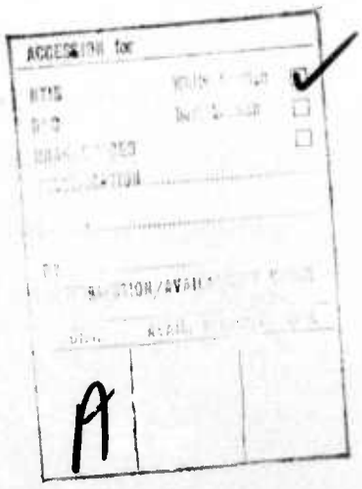
A "majority consensus" algorithm which represents a new solution to the update synchronization problem for multiple copy data bases is presented. The algorithm embodies distributed control and can function effectively in the presence of communication and data base site outages. The correctness of the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

Key Words and Phrases: distributed data bases, distributed computation, distributed control, computer networks, update synchronization, clock synchronization, multiprocess systems.

CR Categories: 4.3, 4.32, 4.33, 4.39, 5.29.

The research reported in this paper was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N0014-75-0773.

* 50 Moulton St., Cambridge, Mass., 02138.



1. INTRODUCTION

In a computer network environment it is often desirable to store copies of the same data base at a number of different network sites. A number of advantages can result from maintaining such duplicate data bases. 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 - data base queries initiated at sites where the data is stored can be satisfied directly without incurring network transmission delays and those initiated from sites "near" the data base sites can be satisfied with less delay than those "farther" from the data base sites; load sharing - the computational load of responding to queries can be distributed among a number of data base 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 data bases 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 maintain and the sites at which to maintain them; selecting a data base site to satisfy a

query request when it is initiated; etc.) are not considered in this paper.

The inherent communicatio delay between sites that maintain copies of a data base makes it impossible to insure that all copies remain identical at all times when update requests are being processed. The goal of an update mechanism is to guarantee that updates get incorporated into the data base copies in a way that preserves their mutual consistency. By this we mean that all copies converge to the same state and would be identical should update activity cease.

Traditional update mechanisms can be characterized as involving some form of centralized control whereby all update requests are channeled through a single central point. At that point the requests can be validated and then distributed to the various data base sites for entry into the data base copies. A second, fundamentally different approach to the update problem, based on distributed control, is possible. For this approach the responsibility for validating update requests and entering them into the data base copies is distributed among the collection of data base 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, might lead to inconsistencies and eventual divergence of the data base copies. The primary disadvantage of

such mechanisms is that data base update activity must be suspended whenever the central control point is inaccessible. Such inaccessiblity could result from outages in the communcations network or of the network 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 which is capable of processing data base updates even when one or more of the component sites are inaccessible (1). The problem here is that it is non-trivial to design a distributed update control mechanism which operates correctly; that is, which can resolve conflicting updates in a way that preserves consistency of the data base copies and is deadlock free. Centralized update control is adequate for many applications. However, there are data base applications whose update performance requirements can be satisfied only by a system which uses distributed update control.

The mechanism for maintaining multiple copy data bases presented in this paper is one which uses distributed control. In particular, the update algorithm presented has the following properties:

. Distributed Updating.
  Updates to a redundantly maintained data base can be initiated through any of the data base sites.

---

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

- Update Synchronization.
  Races between conflicting, "concurrent" update requests are resolved in a manner that maintains both the internal consistency and the mutual consistency of the data base copies.

- Deadlock Prevention.
  The synchronization mechanism that resolves races does not introduce the possiblity of so called "deadly embrace" or deadlock situations.

- Robustness.
  The data base update algorithm can recover from and function effectively in the presence of communication system and data base site failures. The algorithm is robust with respect to lost and duplicate messages, the (temporary) inability of data base managing processes to communicate with one another (due to network or host outages), and the loss of memory (state information) by one or more of the data base managing processes. In developing the algorithm, any mechanism that required all data base managing processes to be up and accessible in order for it to function effectively was rejected. Mechanisms were sought that required only pairwise interactions among the data base managing processes.

- Correctness.
  It is possible to make a strong plausibility argument, which serves as an informal proof, for the correctness of the algorithm.

The remainder of this paper describes the update algorithm. First the algorithm is described in overview and then in detail. Following that, an informal proof for its correctness is presented. Next, the cost of using the algorithm is investigated. Finally, the impact memory loss at data base sites has on the operation of the algorithm is briefly discussed.

2. THE UPDATE ALGORITHM IN OVERVIEW

We assume an environment within which copies of a data base are accessible at a number of data base sites. It is further assumed that the data base copy at each site is acccessible only through a data base manager process (DBMP) which resides at that site. Query and update access to the data base is initiated by application processes (APs). Each access to the data base is completed by a DBMP acting on behalf of the initiating AP.

To query the data base an AP sends a query request to a DBMP. The DBMP acts upon the request by querying its copy of the data base and returning the results to the requesting AP. An interprocess communication facility to support AP-DBMP and DBMP-DBMP communication is assumed. It is further assumed that the facility supports both intra-host (an AP and DBMP may reside at the same network host) and inter-host communication.

We assume that, in general, APs initiate updates by first performing a computation based upon data base values obtained by one or more data base queries, and then submitting an update request to a DBMP. The manner in which a DBMP acts upon an update request is somewhat more involved than that for a query request. As noted earlier, due to the delay inherent in communication between DBMPs, it is not possible to guarantee that the data bases are identical at all times. The objective of the DBMP in processing an update request, therefore, is to maintain both the mutual consistency of the collection of data base copies and the internal consistency of each copy.

By "mutual consistency" we mean that given a cessation of update activity, and sufficient time for each DBMP to communicate with every other DBMP, the data base copies will be identical. The notion of "internal consistency" is somewhat more difficult to define precisely. It has to do with the preservation of invariant relations that exist among items within the data base. As such, internal consistency is related to the interpretation or semantics of items in the data base. Therefore, most of the responsibility for the internal consistency of a data base must rest with the application processes. The DBMPs should be required to know little, if anything, about the data base semantics. The DBMPs should, however, make it possible for a set of well behaved APs to update the data base in a way that preserves internal data relationships.

After an AP initiates an update request, the collection of DBMPs act cooperatively to perform the requested update and notify the AP of its acceptance or rejection. An AP process is free to resubmit a rejected request for reconsideration by the DBMPs.

The DBMPs determine whether to accept a given update request by voting on it. A request that receives a majority consensus from the DBMPs will be accepted. Occassionally a DBMP set must reject requests in order to maintain the consistency of the data base copies. In such a case, a single dissenting vote is sufficient to cause a request to be rejected (see Assertion 3 in Section 4).

As an example, consider a 3 DBMP system for a data base which includes the variables x and y, and assume that x and y have the values 1 and 2, respectively, in all three copies. Suppose that two application processes concurrently request x:=y and y:=x, respectively, by initiating update requests at different DBMPs. After the two updates are completed, one would expect x and y to be equal, although one could not predict whether their value would be 1 or 2. If both requests were to be accepted, x and y would not be equal. Hence, one of the requests must be rejected in order to maintain the (internal) consistency of the data base. Stated somewhat differently, the update request that gets rejected must be refused because it is based on information made obsolete by the request that gets accepted. The AP whose request is rejected is free to resubmit it. If the request is based on current information when it is resubmitted, it can be accepted.

3. THE UPDATE ALGORITHM IN DETAIL

The basis of the distributed update algorithm is the voting procedure used by the DBMPs to determine the acceptability of data base update requests. This section describes the algorithm in detail. It begins by specifying the nature of update requests and the voting rules used by the DBMPs. Next, it discusses the role of timestamps in the algorithm. Finally, the properties of the algorithm which make it robust with respect to communication and data base site outages are described.

3.1 Update Requests and Voting Rules

When an AP submits an update request to a DBMP it includes as part of the request:

- a list of variables to be updated (called the "update" variables) and their new values;
- a list of the variables upon which the update is based (called the "base" variables);
- a list of timestamps for the base variables which indicate when those variables were last updated.

The algorithm requires that the update variables be a subset of the base variables. The reason for this requirement is discussed in section 4.

Typically an AP would accomplish an update by performing the following sequence:

- request base variables and time stamps from a DBMP;
- compute new values for the update variables;
- submit update variables, base variables, and base variable timestamps to a DBMP as an update request;
- if the request is rejected and the update is still desired, repeat this sequence.

When it is requested to vote on an update request, a DBMP may vote "reject" (REJ), "OK", or "deadlock reject" (DR); or, it may (temporarily) defer voting on the request. The DBMP voting rules are:

- Vote REJ only if one or more of the base variable timestamps is obsolete;
- Vote OK only if the base variable timestamps are current and the request is not in conflict with any requests that are currently pending at that DBMP. Two requests are said to conflict if the intersection of the update variables of one request and the base variables of the other request is

non-empty. When a DBMP votes OK on a request, the request is said to be pending at that DBMP.

- Vote DR only if the base variable timestamps are current and the request conflicts with a pending request of higher priority. Each request is assigned a priority which is a function of the DBMP at which it was initiated. (For simplicity assume all requests initiated at the same DBMP have the same priority and that requests initiated at different DBMPs have different priorities.)

- Defer voting only if (a) the base variable timestamps are current, the request conflicts with a pending request, and the priority of the request in question is higher than that of the pending request, or (b) the base variable timestamps are more current than the corresponding data base timestamps. Each DBMP maintains requests that it has deferred in a queue (FIFO).

The algorithm insures that requests upon which voting has been deferred will be automatically reconsidered at a later time after the conflicting, pending request which caused the deferral has been resolved. It can be shown that this reconsideration will occur at an indefinite, but finite future time (see Assertion 6 in section 4).

The voting rules prevent a DBMP from changing its vote on a request if it is given an opportunity to vote again on it. (The mechanisms which insure the algorithm's robustness may cause a DBMP to be requested to vote more than once on an update request.)

A DBMP that votes REJ on an update request is responsible for seeing that the request is properly "rejected" by the set of DBMPs. To do this, it must:

- Notify each other DBMP that the request has been rejected; and

. Inform the requesting AP that the request has been rejected.

When a DBMP discovers that a request has been rejected, it should attempt to vote on those requests which were deferred because they were in conflict with the rejected request. The voting rules specified above are to be used and deferred requests are to be reconsidered starting from the beginning of the queue of deferred requests.

When a DBMP notes that a majority consensus has been reached on a request (i.e., a majority of DBMPs have voted OK), the DBMP must "accept" the update. To accept an update, a DBMP must:

. Update its local copy of the data base as specified by the request update variables;

. Notify each other DBMP that the request has been accepted; and

. Notify the requesting AP that the request has been accepted.

When a DBMP is notified that a request has been accepted it must update its local copy of the data base as specified by the request update variables. In addition, it should reject any pending or deferred requests that conflict with the accepted request since those requests can never gain a majority consensus (1).

---

1. This procedure of rejecting pending requests in conflict may cause a request to be rejected by more than a single DBMP. As we shall see, this causes no problems. Alternatively, a DBMP could reject only deferred requests that are in conflict with the accepted request. Rejecting pending requests results in

The reasoning which lead to the introduction of deferred and DR voting is somewhat involved. The algorithm would be considerably simpler if a DBMP could vote REJ under the DR and deferred conditions. There are two reasons why a DBMP should not vote REJ under these conditions. First, consider a request (U) that would be accepted by the DBMP set in the absence of a pending conflicting request (C). In the case that C is eventually rejected by the DBMP set, by voting REJ for U, the DBMP set would reject U when it need not. The second problem is a more serious one. It has to do with the manner in which the voting rules interact with the mechanisms (to be introduced in Section 3.3) which insure robust behaviour in the presence of communication and DBMP outages. There are outage patterns which, if a DBMP were to vote REJ under the DR and deferred conditions, could result in both acceptance and rejection of a given update request by the DBMP set. Deferring the vote on a request that is in conflict with another pending one addresses both problems at the expense of introducing a potential for deadlocks. Introduction of the DR voting rule insures that deadlocks can not occur (see Assertion 7 in section 4).

A DBMP votes DR only when the variables in an update request conflict with those of another pending request. The intent of a DR vote is to inform other DBMPs that a potential deadlock situation with respect to the request exists. The request in

---

quicker rejection of requests that are destined to be rejected at the expense of introducing possible multiple rejections of the same request.

question can continue to be considered by other DBMPs until sufficient DRs accumulate to prevent a majority consensus on the request. When (if) this condition occurs, the DBMP that detects it must reject the request. In effect, this rejection condition represents a consensus among the DBMPs that the request should be rejected to prevent a possible deadlock. To see the kind of situation the DR rejection rule prevents, consider a 2N DBMP system for which two conflicting update requests are initiated at different DBMPs. It is possible for each request to progress to the point where each has N OK votes. At that point, without a rule such as the DR rejection rule, neither could achieve a majority consensus and a deadlock would result.

If, after voting on an update request, the outcome of the request is still unresolved (i.e., the request base variables are current but there are insufficient OK votes for acceptance or DR votes for rejection), the DBMP should forward the request to some other DBMP which has not yet voted on the request.

The APPENDIX to this paper presents several examples which illustrate how update requests submitted to a DBMP set proceed toward resolution under the voting rules just described.

3.2 Timestamps

Each modifiable data item in the data base has a timestamp associated with it. The timestamp reflects the time at which the item was assigned its present value.

Timestamps are used by the DBMPs in two ways. They are used in the voting procedure. A DBMP can use timestamps to determine whether the base variables of an update request are current by comparing the request base variable timestamps with the corresponding timestamps in its copy of the data base. If any data base timestamp is more recent than the corresponding base variable timestamp, the request base variables are obsolete and the request must be rejected (1).

The second way timestamps are used is to insure that accepted updates are "properly" sequenced as they are incorporated into the data base copies. The manner in which updates are accepted by and communicated among the DBMP set makes it possible for notification of the acceptance of an update (U2) to a data item to arrive at some DBMP before notification of the acceptance of a previous update (U1) to the same item.

For example, consider a 3 DBMP system where DBMP 3 is down when U1 and U2 are accepted. Further, suppose that U1 is initiated by an AP at DBMP 1 and accepted at DBMP 2, and that U2 is initiated later at DBMP 2 and accepted at DBMP 1. Now, assume that when DBMP 3 comes up DBMP 2 is down. DBMP 3 will receive

---

1. Alternatively, a DBMP could determine the currency of request base variables by comparing their values with those in the data base. However, for items which are complex, such as lists or other data structures, the comparison (equality check) could be quite expensive. The cost of the timestamp check is independent of the complexity of the item. We note, however, that the timestamp check is sometimes too strong a condition and could result in rejection of a request that need not be rejected.

notification of U2's acceptance from DBMP 1; sometime later, when DBMP 2 comes up, DBMP 3 will receive notification of U1's acceptance from DBMP 2.

By associating timestamps with update requests it is possible for a DBMP receiving notification of the acceptance of a request to determine the currency of the request. In the case of the example above, when DBMP 3 receives notification of U1's acceptance, it should compare U1's timestamp with that associated with the data item in its copy of the data base. If U1's timestamp is more recent, DBMP 3 should perform the accepted update; otherwise, it should discard U1 as obsolete (1).

The question arises as to when and by whom an update request should be timestamped. There seem to be only two logical choices:

- By the initiating DBMP.
  At the time the update is requested it is timestamped by the DBMP that receives the request from an AP; or
- By the accepting DBMP.
  At the time the update is accepted it is timestamped by the accepting DBMP.

The techniques used to insure robust behavior (described in section 3.3) make it possible for a given update request to be accepted by more than a single DBMP. Therefore, in order to insure a single, unique timestamp, requests are timestamped by the DBMP with which the request is initiated.

---

1. For a request with more than a single update variable, it may be the case that some of the updates to individual variables are performed while others must be rejected as obsolete.

Generating timetamps is a problem. We assume that each DBMP has access to a local, monotonically increasing clock, but that there is no common clock accessible to all DBMPs. Since timestamps are being used to sequence update requests, it is important that no two conflicting update requests have the same timestamp. It is not difficult to insure that two timestamps generated by a given DBMP are unique. To prevent duplication of timestamps generated by different DBMPs, we assume that the low order digit (or digits) of the timestamp obtained from a local clock is unique to each DBMP.

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 [1]. In terms of the previous example, anomalous behavior could result if the timestamp generated by DBMP 1 for U1 is more recent than that generated for U2 by DBMP 2; the anomaly here would be that U1, the earlier update, would be retained in the data base. We shall call such an occurrence a "sequencing anomaly". Such behavior appears anomalous only to an observer (such as a human user or an AP) who can determine by some means external to the DBMP system that U2 "occurred after" U1. However, since a DBMP system functions for such external observers, it is important to prevent sequencing anomalies. Section 4 describes a procedure for choosing timestamps to prevent them (see Assertion 5).

3.3 Robustness

Several observations can be made regarding the robustness of the majority consensus algorithm as it has been described so far.

Only a majority of the DBMPs or fewer (in the case of a rejection) are necessary for an update request to be resolved. Therefore, the data base can undergo modification when some DBMPs are inaccessible. Furthermore, the majority of DBMPs necessary for a consensus need not all be available at the same time. Since the algorithm involves only pairwise interactions among processes, an update request can advance toward a consensus or rejection when only two DBMPs are up.

It is not necessary for the DBMP at which a request is initiated to remain up in order 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.

As the mechanism has been described, progress toward the resolution of an update request can temporarily cease only if:

- a DBMP that is trying to forward an unresolved request is unable to find another DBMP that is accessible and has not yet voted on the request.
- a DBMP that is trying to forward an unresolved request crashes before it is able to forward the request.

In first case there is little the DBMP can do 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 non-voting DBMP becomes accessible (1).

The use of timeouts can make the data base mechanism robust with respect to failures of the second type. A DBMP which has successfully forwarded an unresolved update request should time the request out in the following sense. If the DBMP does not hear that the request has been resolved within a timeout period it should act to help the request progress further toward resolution.

A procedure that a DBMP can use when a request is timed out is to check the status of the DBMP (call it X) to which it forwarded the request. If X is not up, then the checking DBMP 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 the same procedure is used by X to insure that the request proceeds toward resolution

---

1. We assume that the DBMPs are persistent with respect to all the messages they send to each other and to APs. That is, the DBMPs cooperate to implement a reliable message transmission mechanism. The critical property of such a mechanism is that the delivery of interprocess messages is guaranteed even if the receiving process is inaccessible when the sending process initiates the message transmission. We note that the "network mail" facility of the ARPANET [2] incorporates such a reliable transmission mechanism to insure that network mail is always eventually delivered. The details of how such mechanisms can be implemented, though important, will not be discussed here.

(2).

This procedure is analogous to the "timeout and retransmit" procedures used in many network communication protocols [3]. The procedure contributes to the robustness of the data base 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.

A side effect of this retransmission procedure is that a DBMP may be asked to consider a given update request more than once. This is similar to the receipt of duplicate messages in a communication system which uses retransmission. Duplicate requests represent no problem as long as a DBMP can determine whether it has already voted on a request it is asked to consider, and, if it has, that it does not change its vote.

A second (or third, etc.) request to consider a given update may traverse a different path through the network of DBMPs than the first. As a result, such a request may provide the receiving DBMP with new information regarding the status of the update request. That is, when the votes on the duplicate request are merged with the votes already known to the DBMP there may be sufficient OK votes for a consensus or sufficient DR votes for a

---

1. 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.

rejection. Or, the receiving DBMP may detect that, althougn the request remains unresolved, the number of OK and/or DR votes has increased.

3.4 IDs for Update Requests

The data base algorithm requires that update requests be uniquely identified within the set of DBMPs. This ID is used in a number of ways. When voting on an update request, a DBMP must be able to determine whether it has previously voted on the request. Similarly, in order for DBMPs to be able to "garbage collect" storage used for maintaining state information for pending requests, when a DBMP is informed of the resolution of a request, it must be able to determine whether it has any record of the request.

The initiating DBMP is the process responsible for generating unique IDs and associating them with updates requested by APs. We note that the update timestamp generated by the initiating DBMP for a request is unique and, therefore, is adequate to serve as a request ID.

Should an update requested by an AP be rejected, subsequent requests by the AP to accomplish the "same" update are regarded by the DBMP set as different requests. That is, each request is given a unique ID when it is submitted to the DBMP set for consideration.

4. ALGORITHM CORRECTNESS

This section presents plausibility arguments which serve to explain how and why the majority consensus algorithm works. Taken together, these arguments represent an informal proof for the correctness of the algorithm.

The important aspects of the algorithm operation are the use of timestamps in the voting procedure, the relationship between the base and update variables in update requests, and the assumed reliable transmission mechanism. The reliable transmission mechanism guarantees that inter-DBMP messages are always (eventually) delivered. The comparison of update timestamps with data base variable timestamps made at each DBMP when an accepted update is performed, together with the transmission mechanism, guarantees mutual consistency. In effect, for each item in the data base, each DBMP is able to reconstruct and then act upon the same sequence of update events as each other DBMP.

The base variables of update requests are intended to be used by APs and DBMPs as an aid for insuring internal data base consistency. The intent is that an AP specify the base variables which represent the premises upon which an update request is based. The timestamp check for the request base variables and the check for conflicts with pending requests made as a DBMP votes on an update request insures that the premises upon which the requesting AP has based the update have not changed. Including the update variables in the base variable set is, in

effect, equivalent to the premise that the requesting AP is the only process updating the variables in question.

Definitions and assertions about the update algorithm constitute the remainder of this section.

Definition (Cover):

> An update request A is said to "cover" another update request B if and only if there is at least one variable which is an update variable of A and a base variable of B. That is, A covers B if and only if A's update variables and B's base variables have a non-empty intersection.

The voting rules prevent a DBMP that has voted OK on a request A, which to its knowledge has not yet been resolved, from voting OK on any request covered by A.

Definition (Concurrent):

> Two requests A and B are said to be "concurrent" if and only if each variable v in the intersection of their base variables has the same timestamp in request A and in request B.

Assertion 1:

> An update request that is covered by another, concurrent request which has been accepted by the set of DBMPs can not be accepted.

Plausibility Argument:

Let A be the accepted update and B be the update under consideration.

Assume that B is accepted by the set of DBMPs. A majority of DBMPs must have voted OK on B. Similarly, a majority of DBMPs must have voted OK on A. Therefore, at least 1 DBMP, call it X, must have voted OK on both A and B.

When X voted OK on B either A was pending at X (because X had not yet heard of A's acceptance) or A had been performed by X (1). If A was pending, X could not vote OK on B since A covers B and the voting rules prevent such a vote. If A had been performed, X could not vote OK on B because at least one of B's base variable timestamps would be obsolete since at least one of B's base variables is an update variable of A (A covers B). Therefore, X could not have voted OK on B. This is a contradiction.

Therefore, the assumption that B is accepted is false and the assertion is true.

Definition (Conflict):

Two update requests A and B are said to "conflict" if A covers B or if B covers A.

---

1. The case that X has not yet heard about A need not be considered since it is assumed that A has been accepted and that X is one of the DBMPs that voted on A.

Assertion 2:

When the set of DBMPs considers two conflicting, concurrent update requests, at most one of the requests will be accepted.

Plausibility Argument:

The reasoning here is similar to that for Assertion 1.
Let A and B be the two updates.

Assume that both A and B are accepted. Because each must accumulate a majority consensus, there must be at least 1 DBMP which votes OK on both A and B. However, there can be no such DBMP since the voting rules prevent a DBMP from voting OK on conflicting requests. Therefore both A and B cannot both be accepted.

Assertion 3:

If a single DBMP rejects an update request, U, it is not possible for U to achieve a majority consensus.
(That is, even if all other DBMPs were to be given an opportunity to vote on U, U would not receive a majority consensus.)

Plausibility Argument:.

Let X be a DBMP that rejects U.
The voting rules are such that X will reject U only if:

a. U has accumulated sufficient DR votes to prevent a consensus; or

b. U is covered by another request (A) that obsoletes U's base variables and that has already been accepted by the set of DBMPs and performed by X.

Consider case (a). Since the DBMPs that have voted DR may not change their votes U cannot achieve a majority consensus.

Next, consider case (b). Either X voted OK for A or it did not.

First, assume that X did not vote OK for A. For U to be accepted there must be a DBMP different from X in the majority sets of both A and U (since X is not in the majority set of A). Such a DBMP can not exist because the voting rules prevent a DBMP from voting OK for both A and U since A conflicts with U.

Now, assume that X voted OK for A and further, assume that the number, n, of DBMPs is odd:

$$n = 2m - 1 ;$$

a majority of the DBMPs number at least m. In addition to X, at least m-1 other DBMPs voted OK for A and are prevented by the voting rules from voting OK for U. U cannot achieve a majority consensus among the remaining DBMPs that did not vote for A (and number at most m-1). A similar argument holds when n is even.

Thus, the assertion is true.

Assertion 4:

It is not possible for the same request to be both accepted and rejected by the set of DBMPs.

Plausibility Argument:

This assertion follows from Assertion 3.

Assertion 5:

The following function generates timestamps for update requests in a way that prevents sequencing anomalies:

ts = max(time, 1 + max(U.BaseVar.Timestamps))

"time" is the time obtained by a DBMP from its local clock; and

U.BaseVar.Timestamps is the set of timestamps for request U's base variables.

Plausibility 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. We wish to show that the value of any data base variable which is both in A's and B's update variables will be specified by B. Since we assume that A and B have update variables in common, A covers B. We wish to show that if the function above is used to generate the timestamps, Ta and Tb, for A and B, then a sequencing anomaly can not occur. That is, we wish to show that:

Tb > Ta.

Since B is covered by A and we have assumed that B is accepted after A, it must be the case that B is initiated at a DBMP that has performed A; otherwise, the timestamps of at least one of B's base variables would be obsolete, leading to B's eventual rejection. Because A covers B, there is at least 1 variable, v, that is a base variable of B and an update variable of A. The timestamp of v is Ta. The function guarantees that Tb is at least Ta + 1. Therefore, Tb > Ta, under the stated assumptions.

Assertion 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. The only requirement is that local DBMP time never back up. Assertion 5 is an important result because it is very difficult to synchronize clocks in a distributed environment.

We note that it does not follow from this assertion that any two events initiated at different DBMPs in a system with asynchronous DBMP clocks can be properly sequenced. It only insures 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 is referred to [4].

Assertion 6:

An update request U 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.

Plausibility Argument:

Let U be initiated at DBMP I. I has 4 options with respect to U:

1. it can reject U;
2. it can vote OK on U;
3. it can vote DR on U; or
4. it can defer voting on U.

If DBMP I rejects U (case 1), U is resolved (in finite time). Consider cases (2) and (3). After voting, I can forward U to another DBMP J that has not voted on U. Our premise assures that this is done in finite time.

DBMP J has the same 4 options with respect to U. If it rejects U, U is resolved in finite time. If it votes OK or DR and there are insufficient votes to resolve U (neither

enough OKs for a consensus nor DRs to prevent a consensus), J will forward U to another DBMP K that has not yet voted on U, thereby, in finite time, advancing U one step closer toward resolution. Since there are at most n (= number of DBMPs) such steps required for U's resolution, it suffices to show that each step requires only finite time.

The only case that is potentially troublesome is when a DBMP defers voting on U. A DBMP K will defer voting on U only if U conflicts with a pending request (L1) of lower priority. The voting rules then prevent K from considering U until L1 is resolved. If L1 is resolved, then K will learn of the resolution in finite time. If L1 is accepted, K will reject U; if L1 is rejected then K may vote on U. K's vote will result either in U's resolution or the advancement of U one step further toward resolution. Therefore, if the request L1 that caused U to be deferred is resolved within finite time, then U will either be resolved or advanced one step further toward resolution by DBMP K in finite time.

We turn our attention now to L1. L1 will be handled by the DBMP set similarly to U. That is, it will proceed toward resolution at a finite rate unless (and until) it is deferred by some DBMP because it conflicts with a pending request of lower priority.

Hence, progress toward U's resolution may be blocked by a finite chain of requests L1,L2, ..., Lr; where U is deferred

at some DBMP K because it conflicts with the lower priority request L1; L1 is deferred at some other DBMP because it conflicts with lower priority request L2; etc.

The lowest priority request Lr will be resolved in finite time because the voting rules prevent it from being deferred by any DBMP. The voting rules require that the chain of deferred requests be reconsidered in a FIFO manner. Therefore, when Lr is resolved, either Lr-1 will be rejected or it will be advanced one step further toward resolution. Hence, Lr-1 will proceed toward resolution at a finite rate and therefore will be resolved in finite time, enabling Lr-2 to be resolved in finite time, ... enabling L1 to be resolved in finite time. Therefore, U will be resolved in finite time.

Assertion 6 is an important result. From it, it follows that:

Assertion 7:

The DBMP set is deadlock free.

A guaranteed finite time for pairwise DBMP communication is a necessary condition of Assertion 6 because at any given time communication between a given pair of DBMPs may not be possible due to network or host failure or outage. It is possible that the outages and recoveries occur in such a way as to prevent a

request from ever being resolved. For example, consider a two DBMP system in which the DBMPs are never up at the same time; e.g., DBMP 1 is up from time 0 to T-e, 2T+e to 3T-e, ..., 2mT+e to (2m+1)T-e, ... and DBMP 2 is up from T+e to 2t-e, ..., (2m-1)T+e to 2mT-e, ... Clearly, no request can ever be accepted by such a system because DBMPs 1 and 2 can 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.

5. COST OF THE ALGORITHM

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

- Communication. A number of interprocess messages must be exchanged to accomplish an update;
- Computation. The update must be computed. This requires one or more queries to obtain the base variables and computation of the values for the update variables. The race resolution mechanism occassionly 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.
- Delay. It takes some time for the DBMP set to resolve an update request.

This section examines the communication and computation costs imposed by the algorithm.

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	For AP to initiate the update (AP->DBMP messages to request variables, transmit variables, request update)
+ n/2	To achieve a consensus (inter-DBMP messages);
+ n-1	To notify the DBMP set of acceptance (inter-DBMP messages);
+ 1	To notify AP of acceptance (DBMP->AP message)

or $n + (n/2) + 3$

messages.

If there are conflicts, the votes of more than n/2 DBMPs may be required to resolve a request. Each additional DBMP vote requires an additional message. In the worst case, every DBMP would have to vote before a request could be accepted. This would require n-1 inter-DBMP messages. Therefore, in the worst case, a request would require

$2n + 2$

messages to be accepted.

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 $n + (n/2) + 3$ compares favorably with other techniques one might consider for managing distributed, redundant data bases.

An update algorithm is described in [1] which guarantees mutual consistency but can not insure internal consistency of data base 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 initiatng DBMP to acknowledge
            the update;
+ n-1      To communicate the update to the other DBMPs
```

or

$$n + 3$$

messages. The difference of (n/2) 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 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 3 messages to intitiate the update; n-1 messages are required to distribute the update to the other DBMPs; and 1 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 data base 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 data base;
+	1	To obtain the base variables;
+	n	To perform the update and unlock the data base copies

or

$$2n + 1$$

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

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

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

$$C + (n+n/2+3)M \leq CO \leq C + 2(n+1)M.$$

If we define

$$COmin = C + (n+n/2+3)M$$
$$COmax = C + 2(n+1)M$$

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

$$COmin + C + 2M \leq C1 \leq COmax + C + 2nM$$

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 (1); 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 resubmitted to the DBMP set k times before it is accomplished is:

$$COmin + k(C + 2M) \leq Ck \leq COmax + k(C + 2nM)$$

---

1. This assumes that 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.

6. THE PROBLEM OF MEMORY LOSS

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

If a DBMP that has lost memory is permitted to vote on update requests, that DBMP can cause the majority consensus algorithm to malfunction. This can happen if: (1) the DBMP votes OK for a request which conflicts with accepted updates it has forgotten, thereby possibly enabling the request, which it should reject, 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 DR), 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 catastropic event at the data base 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 presumably 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 signalled in some way 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 non-volatile storage medium and that this record is archived along with the data base and also restored whenever 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 requests it receives from other DBMPs. In order to become a voting member of the DBMP set, a DBMP that has lost memory must:

- 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);
- 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 (1).

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 (2).

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.

---

1. 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.

2. This temporary freezing of data base activity with respect to these unresolved requests prevents the pathological behavoir mentioned in the previous footnote.

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 data bases. Because the responsibility for performing an update is distributed among the collection of processes that manage data base copies rather than centralized in a single process, the algorithm can function effectively (i.e., process updates) in the presence of communication and data base 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 traditional 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 data base 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 expected patterns of update activity. In practice, the dominant cost associated with use of the algorithm is likely to be that incurred to satisfy this short term memory requirement.

A multiple copy data base is one particular type of distributed data base. Another type is one which consists of distributed, non-overlapping segments; that is, a data base which is a collection of smaller data base segments each of which is singly maintained at a (possibly) different site (1). Although the data itself is not redundantly stored for this type of distributed data base, in some applications it may be desirable to maintain multiple copies of the catalogues for such a segmented data base. For these applications the majority consensus algorithm could be used to handle updates to the data base catalogue.

---

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

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

- How should application processes be programmed to deal with the fact that data found in any given data base copy may not be the most current? In some cases it may not be critical that the data is not current. If it is critical, how can a process locate the most current data?

- 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?

- 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 outages), a simpler, centralized recovery procedure may be adequate in most situations.

ACKNOWLEDGEMENTS

The notion that a voting procedure might form the basis for achieving update synchronization is due to Leslie Lamport. Paul Johnson and Harry Forsdick contributed to the formulation of the ideas in this paper. In addition, conversations with Rick Schantz, Jerry Burchfiel, and Ray Tomlinson were helpful.

REFERENCES

[1] Johnson, P. and Thomas, R., The Maintenance of Duplicate Data Bases, ARPA Network Working Group Request for Comments (RFC) #677, Network Information Center (NIC) Document #31507, January 1975.

[2] Kimbleton, S. and Schneider, G., Computer Communication Networks: Approaches, Objectives, and Performance, ACM Computing Surveys, Vol. 7, No. 3, September 1975, pp. 129-173.

[3] Cerf, V. and Kahn, R., A Protocol for Packet Network Interconnection, IEEE Transactions on Communications, Vol Comm-22, No. 5, May 1974, pp. 637-648.

[4] Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, Massachusetts Computer Associates Report # CA-7603-2911, March 1976; also submitted to Communications of the ACM.

APPENDIX

This appendix includes a number of examples chosen to illustrate various aspects of the update algorithm.

Before presenting the examples 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).
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 data base.
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 means that DBMPs 1 and 2 have voted OK 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.

Example # 1: Normal update with no conflict.

Consider 3 DBMPs which manage a data base 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 data base, 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 table below illustrates the sequence of events that occur and how the status of the request A as seen by each DBMP evolves as the DBMPs work to accomplish the update.

	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 do@2	*DONE do@2	--

[2->3:DO(A)]

A	DONE do@2	DONE do@2	DONE do@2

[1,2,3 discard (1) request A]

Example # 2: Concurrent Conflicting Updates.

This is the example from section 2. There are 3 DBMPs which manage a data base that includes variables x and y. Assume all data bases are current and x=1 and y=2 in all copies of the data base. Assume that AP1 initiates update A and that AP2 initiates update B:

A: x := y
B: y := x.

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

In the following, A is accepted causing x to be set to 2 and B to be rejected. AP2 then chooses to re-initiate its

---

1. A DBMP may "discard" an accepted update request after it has entered the update into its data base 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.

update (called B' to distinguish it from the AP2's original request) which updates y to 2. We assume that the priority of a request initiated at DBMP 1 is greater than that of one initiated at DBMP 2 or 3, etc.

	DBMP-1	DBMP-2	DBMP-3
Status of:			
A	--	--	--
B	--	--	--
[AP1->1:RV(xy) & AP2->3:RV(xy) / 1->AP1:VAR(xy) & 3->AP2:VAR(xy) / 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-DR]			
A	ok@1	*ok@12	--
B	*ok@3dr@1	--	ok@3
[2 accepts A & 1->2:RC(B) / 2->AP1:UA(A) & 2->1:DO(A) & 2 rejects B]			
A	DONE do@2	*DONE do@2	-- --
B	ok@3dr@1	*REJD rej@2	ok@3
[2->3:DO(A) & 2->1,3:REJ(B) & 2->AP2:UR(B,x,y)]			
A	DONE do@2	DONE do@2	DONE do@2
B	REJD rej@2	REJD rej@2	REJD rej@2
[1,2,3 discard A and B]			
A	--	--	--
B	--	--	--
[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 3 DBMPs which manage a data base which includes the variables x, y, and z. Assume that all copies of the data base are current and that x=1, y=2, and z=3. Assume that 3 application programs attempt the updates:

```
A: x := y*z   (by AP1)
B: y := z + x (by AP2)
C: z := x - y (by AP3)
```

Update A would change x to 6; B would change y to 4; C would change Z to -1. The base variables of all 3 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.

	DBMP-1	DBMP-2	DBMP-3
Status of:			
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-DR]

```
A          ok@1                 DEFR,ok@1            --
B          --                   ok@2                 DEFR,ok@2
C          *ok@3dr@1            --                   ok@3

[ 1->2:RC(C) / 2 votes C-DR / 2 rejects C ]

A          ok@1                 DEFR,ok@1            --
B          --                   ok@2                 DEFR,ok@2
C          ok@3dr@1             *REJD                ok@3
                                rej@2

[ 2->1,3:REJ(C) & 2->AP3:UR(C,x,y,z) / 3 votes B-OK / 3 accepts B ]

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

[ 3->1,2:DO(B) & 3->AP2:UA(B) / 1,2,3 discard C & 1,2 reject A ]

A          *REJD                *REJD                --
           rej@1                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,x,y,z) &
  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 5 DBMP system and that all data base 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-DR / 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@3dr@1	*ok@23dr@1	ok@3	XX	XX

[2 crashes / 1 times out A]

A	*ok@1	XX	--	XX	XX
B	ok@3dr@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@3dr@1	XX	*ok@3	--	--

[3->4:RC(B) / 4 votes B-OK]

A	ok@1	XX	DEFR,ok@1	--	--
B	ok@3dr@1	XX	ok@3	*ok@34	--

[3,4 crash / 1 times out A]

A	*ok@1	XX	XX	XX	--
B	ok@3dr@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@3dr@1	*ok@23dr@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-DR]

A	ok@1	DEFR,ok@1	DEFR,ok@1	DEFR,ok@15	ok@15
B	ok@3dr@1	*ok@23dr@1	ok@3	ok@34	*ok@34DR@5

[2->5:RC(B) / 5 accepts B / 5->1,2,3,4:DO(B) & 5->AP2:UA(B) / 5 rejects A & 1,2,3,4 reject A]

A	*REJD	*REJD	*REJD	*REJD	*REJD
	rej@1	rej@2	rej@3	rej@4	rej@5
B	DONE	DONE	DONE	DONE	*DONE
	do@5	do@5	do@5	do@5	do@5

[1,2,3,4,5 exchange REJs for A / 1,2,3,4,5 discard A,B]

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN Report No. 3340
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A SOLUTION TO THE UPDATE PROBLEM FOR MULTIPLE COPY DATA BASES WHICH USES DISTRIBUTED CONTROL
5. TYPE OF REPORT & PERIOD COVERED: Technical Rept.
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): R. Thomas
8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0773, ARPA Order-2935
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, Massachusetts 02155
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: BBN-3340
12. REPORT DATE: July 1976
13. NUMBER OF PAGES: 50
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This research was supported by the Defense Advanced Research Projects Agency under ARPA Order No. 2935.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed data bases
distributed computation
distributed control
computer networks
update synchronization
clock synchronization
multiprocess systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A "majority consensus" algorithm which represents a new solution to the update synchronization problem for multiple copy data bases is presented. The algorithm embodies distributed control and can function effectively in the presence of communication and data base 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 an appendix.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

060100