A RELIABLE DISTRIBUTED CONTROL ALGORITHM FOR UPDATING REPLICATED DATABASES*

George Gardarin

Institut de Programmation
4 Place Jussieu
75320 Paris Cedex 05, France

Wesley W. Chu

University of California
Los Angeles, Ca. 90024

Abstract

This paper presents a robust, deadlock-free and distributed control algorithm for consistently updating replicated databases. This algorithm is based on local locking and time stamps on lock tables which permit detection of conflicts among transactions executed at replicated databases. Messages are exchanged in the network whenever a transaction commitment occurs, that is, at the end of every consistent step of local processing. Conflicts among remote transactions are resolved by a roll back procedure. Local restart is based on a journal of locks which provides backup facilities. Performance in terms of the number of messages and volume of control messages of the proposed algorithm is compared with that of the voting and centralized locking algorithms. The results reveal that the proposed distributed control algorithm performs, in most cases, comparably to the centralized locking algorithm and better than the voting algorithm.

1. Introduction

The algorithms proposed for updating replicated databases can be separated into two classes: global locking and time stamping. For the global locking scheme, either centralized control [Mena 78, Ston 78] or distributed control [Elli 77] can be implemented. However, both cases require many control messages. For the timestamping method [Thom 78, Bern 78], timing information permits the ordering of transaction updates. However, the addition of a time stamp to every data element in the database significantly increases the storage cost. To reduce storage and communication costs for updating replicated data bases, we propose the following new algorithm that uses local locking and time stamps on lock tables (the set of record identifiers and the description of the operations performed corresponding to a transaction) rather than on every record. Local locking is utilized to maintain internal consistency of the file copy that is being updated or modified, and time-stamping of lock tables permits enforcement of mutual consistency among the replicated databases.

* This research is supported by IRIA, SIRIUS and the U.S. Office of Naval Research, Contract no. N00014 - 75-C-0650.

Mutual consistency is checked after every consistent step of local processing. In other words, global locking is postponed until the commitment of the transaction is executed. This allows us to group the control messages at the end of a transaction, and thus reduces overhead in control messages.

In this paper, we first present the local locking algorithm that ensures local consistency. Next, we describe the distributed commitment algorithm, the recovery procedure, and the deadlock detection algorithm. Finally, we discuss the performance comparisons of the proposed algorithm with other algorithms.

2. LOCAL LOCKING IN EXECUTING A TRANSACTION

2.1 Transaction Execution

A transaction is defined as a unit of consistent processing; that is, a process or a part of a process which performs one or more operations on the distributed database and satisfies the integrity constraints. Transactions are processed locally at each site. A transaction performs a sequence of operations (i.e., read, update, ...) on parts of the database which are shared among the concurrent processes. Such parts are called resources. Depending on the system, such parts can be pages, records, tuples or files of a database. Before starting to execute an operation for a transaction at a site, the local database manager locks the local resource(s) required by this operation. After the local transaction is completed, a commitment algorithm handling the consistency of the distributed database is performed to update redundant copies of the database at other sites.

2.2 Local Locking Principles

The local locking algorithm involves transactions, resources and operations. Each transaction is identified by a unique time stamp value in the system which is the local time at the start of the transaction together with the local site number [Lamp 78b]. It is possible to lock one or several resources by invoking the LOCK function. The LOCK function takes the value zero if the locks can be granted. If the locks are denied, then the LOCK function gives the time stamp value

which identifies the first transaction preventing the acceptance. There is also an unlock procedure to unlock all resources locked by a transaction.

A lock request consists of three elements: a resource identifier array QID, a mode operation array QOD, and a transaction identifier time stamp t . The resource identifier array QID identifies the list of resources to be locked and the mode operation array QOD gives the corresponding operation modes. There is a lock table at each site for each transaction in execution which consists of two sets of information: a set of resource identifiers RID and a set of operation mode vectors MOD. The i th resource identifier locked by transaction t is denoted $RID(t, i)$; the array $RID(t, *)$ represents all the identifiers of the resources locked by transaction t . Likewise, $MOD(t, i)$ denotes the i th operation mode vector associated with transaction t . In addition, table TRANS gives the list of transactions in execution and points to the corresponding lock tables. Whenever a lock request is granted, QID and QOD are recorded as an entry as RID and MOD in a lock table for the transaction. The organization of a lock table of a site is shown in Figure 1.

Figure 1. The Lock Table Organization of a Site

To define the type of operation requested by a transaction, we utilize an operation mode vector which consists of a set of bits. Bit j is one when the corresponding resource is utilized in mode j ; otherwise, it is zero. For the simplest form, the algorithm supports two operation modes: protected read and update. However, it can be generalized to other operation modes such as dirty read, protected update, exclusive modification [Coda 71], etc. The compatibilities of execution of the operation modes can be defined by a matrix C where element $C(i, j)$ has a value 1 if and only if modes i and j are compatible, and has a value 0 otherwise.

2.3 Lock and Unlock Algorithms

We will now describe the lock algorithm as shown in Procedure 1. The statement of line 5 computes the set of resources both requested by transaction t and already locked by transaction k . The statement of line 7 verifies that the requested operation modes are compatible with the current ones on every resource [Gard 76]. The ADD operation (line 13) updates the content of the lock table for transaction t .

```

1. function LOCK(QID, QOD, t);
2. begin
3.   for each k of TRANS do
4.     if  $t \neq k$  then
5.        $R := QID \& RID(k, *)$ ;
6.       for each r of R do
7.         if not ( $MOD(k, r) \subset \neg(QC * QOD)$ )
8.           then LOCK := k; exit;
9.       fi;
10.    od;
11.  fi;
12. od;
13.  ADD(TRANS, t, QID, QOD);
14.  LOCK := 0;
15. end;
```

Procedure 1: The Lock Algorithm

Symbol: $\neg$ = logical negation
 $\&$ = logical and
 $\subset$ = logical inclusion

The unlock algorithm which unlocks all resources locked by a transaction is described below. The SUPPRESS operation (line 3) suppresses the entry for transaction t from the TRANS array and thus erases the corresponding tables RID and MOD.

```

1. procedure UNLOCK(t);
2. begin
3.   SUPPRESS(TRANS, t);
4. end;
```

Procedure 2: The Unlock Algorithm

3. The Commitment Algorithm With Time Stamps

3.1 Principles of the Commitment

The consistency of a distributed data base is threatened by concurrency conflicts as well as site crashes. The proposed commitment algorithm deals with both of these problems. The algorithm consists of four steps. The first two steps mainly ensure the mutual consistency of databases. During the first step, the lock table and the time stamp of the transaction that requests to commit are broadcasted to the other sites. When a lock table message has been received by another site, a locking algorithm is executed. The locking algorithm grants the locks if no conflicting resource in the local database is locked by a transaction with an older time

stamp value. In addition, all younger transactions (with a more recent time stamp value) holding conflicting resource(s) are canceled (aborted). If the copies of the resources found in the lock table can be locked at every up site, then we perform the second step of the commitment which broadcasts the update message to all the sites.* The remaining two steps of the algorithm (a two step commitment [Lamp 76]) mainly deal with site crashes and transaction cancellation. First, an insurance message is broadcasted by the site that performs the transaction. This insurance message requests other sites to insure the lock table, that is, to prevent cancellation of the transaction by any other transaction with an older time stamp. If the insurance message is accepted by all sites, then the algorithm executes the last step as follows: 1) the lock table is recorded in the journal of the site performing the transaction; 2) a "done" message is broadcasted to all sites by the site performing the transaction; 3) when receiving the "done" message, every site executes a local algorithm which records the lock table in its local journal and unlocks the corresponding resources. In order to ensure that other sites would be aware of the completion of the transaction even when the committing site crashes, actions 1) and 2) are assumed to be an atomic operation, that is, either they are completely performed or not performed at all.

As specified in the next three sections, certain situations, such as failure of the site attempting to commit a transaction, can cause transactions to be aborted. The site that aborts a transaction sends an abort message to all sites to release the locks set by that transaction. Furthermore, the originating site performs the CANCEL operation which rolls back the local updates of the transaction.

3.2 Commitment with Down Sites

We assume that when a message is sent from one site to another, the transport mechanism is able to determine if the destination site is crashed or unable to receive a message (for example, with a time-out and retransmission feature). When a message cannot be sent to a site, that site is considered down. In the following, we assume that a site is in either one of the two states: up or down. A down site is considered as disconnected from the network and does not accept any message. The transport mechanism ensures that an up site always receives the messages which are sent to it. Given the network topology and knowledge of the down sites, we are able to detect network partitioning. Under this condition, to ensure data integrity, updates involving more than one partition should not be performed. The commitment

* An improved algorithm could derive the lock table from the update messages and suppress the first step. However, the update messages do not imply the read locks and using two steps avoids the transmission of update messages when the locking fails.

algorithm can ignore down sites which do not cause network partitioning and commitments are performed on all the up sites (which always have an up-to-date version of the data base).

3.3 Local Commitment Algorithm

The local commitment algorithm is the set of procedures for commitment that are executed by the site originating the transaction. In the local commitment algorithm (Procedure 3), we use the time stamp of the transaction, t , as the parameter of the algorithm. The BROADCAST procedure broadcasts the message defined by its parameters. The first parameter is the name of the message and the rest of the parameters are the content of the message. MONITOR is the name of the local executive. When a message named m with parameters x is received by a site, the algorithm labelled " $m(x)$ " is executed with the parameter values found in the message. When a message cannot be sent to a site for a transaction t (by BROADCAST command, it is assumed that the control is passed to the next section labeled " $site\ down(t)$ " in the algorithm. Every up site must send back either a grant or an abort message after any request. The sequence of operations contained in the double brackets $\ll \gg$ is assumed to be atomic, that is, a sequence either completely executed or not executed at all. The CANCEL procedure is assumed to roll back the local updates of the transaction.

```

1. **procedure COMMIT(t);
2. begin
3.   ** step 1 **
4.   lcount(t) := 0 ;
5.   dcount(t) := 0 ;
6.   call BROADCAST('lockt',t,RID(t,*),MOD(t,*));
7.   call MONITOR;

8.   ==site down(t) ;
9.   dcount(t) := dcount(t)+1 ;

10.  ==message lgrant(t) ;
11.  lcount(t) := lcount(t)+1 ;
12.  if (lcount(t)≠n) then call MONITOR ;
13.  else ** step 2 **
14.    "update messages are broadcasted"
15.    ** step 3 **
16.    icount(t) := 0 ;
17.    dcount(t) := 0 ;
18.    call BROADCAST('insure',t) ;
19.    call MONITOR ;

20.  ==site down(t) ;
21.  dcount(t) := dcount(t)+1

22.  ==message igrant(t) ;
23.  icount(t) := icount(t)+1 ;
24.  if (icount(t)≠n) then call MONITOR ;
25.  else ** step 4 **
26.    <<
27.    call WRITE(t,RID(t,*),MOD(t*),'journal') ;
28.    call BROADCAST('done',t) ;
29.    >>
30.    call UNLOCK(t) ;
31.  fi;

```

```

32. fi;
33. ==message abort(t) ;
34. call CANCEL(t) ;
35. end ;

```

Procedure 3: The Local Commitment Algorithm

Note that the update messages are broadcasted to all the sites before taking the insurance. The order of insurance and updating steps can be reversed. In that case, another control message must be added to realize the two step commitment. Using such a scheme avoids abortion of a transaction whose update messages have already been broadcasted, but increases the number of control messages for a commitment of successful cases. Because transaction conflicts are seldom [Gard 79], it requires less in overhead to optimize the successful cases.

3.4 Remote Commitment Algorithm

Parallel to the COMMIT procedures executed by the site managing the committing transaction, every up site runs a specific algorithm for each type of message it receives as described in Procedure 4. In the remote commitment algorithm, INSURE is a boolean array. Its element *i* is true if transaction *i* has been insured, and false otherwise. SEND is a procedure which sends back a message to the site executing the committing transaction. The content of this message is defined by its parameters (as in the BROADCAST procedure). The LOCK function invoked in the procedure is the local LOCK algorithm described earlier.

When the site performing the transaction fails before the insurance is granted, some sites will detect a down condition and initiate the ABORT procedure. Thus the transaction will be aborted on every site in the system. On the other hand, if the site fails after the insurance has been granted but before executing the atomic operation sequence (logging the lock table into its journal and broadcasting the "done" messages), then all sites in the system could wait indefinitely. To avoid such a condition, each site must insert a delay after sending the "insurance grant" message. If the delay expires before reception of a "done" message, then the "site down" algorithm is executed. The insertion of the delay is performed by executing the DELAY procedure. The ABORT procedure broadcasts an "abort" message to all the sites and invokes the CANCEL procedure.

```

1. ** procedure COCOMMIT ;
2. begin
3.   call MONITOR ;

4.   ==message lockt(t,QID,QOD) ;
5.   i := LOCK(QID,QOD,t) ;
6.   while i≠0 do
7.     if i>t & INSURE(i)=false
8.     then call ABORT(i) ;
9.     else call ABORT(t) ;

```

```

10.    call MONITOR ;
11.   fi ;
12.   i := LOCK(QID,QOD,t) ;
13.   od ;
14.   call SEND('lgrant',t) ;
15.   call MONITOR ;

16.   ==message update(...) ;
17.   "perform the update but stay ready to undo it"
18.   call MONITOR ;

19.   ==message insure(t) ;
20.   if t in TRANS
21.   then INSURE(t) :=true;
22.    call SEND('lgrant',t) ;
23.    call DELAY ;
24.   else call ABORT(t) ;
25.   fi ;
26.   call MONITOR ;

27.   ==message done(t) ;
28.   call WRITE(t,RID(t,*),MOD(t,*),'journal') ;
29.   INSURE(t) :=false ;
30.   call UNLOCK(t) ;
31.   call MONITOR ;

32.   ==site down(t) ;
33.   call ABORT(t) ;

34.   ==message abort(t) ;
35.   call CANCEL(t) ;
36.   end;

```

Procedure 4: The Remote Commitment Algorithm

3.5 Proof of Consistency

In order to prove the consistency of the proposed algorithm, we must show that the result of any simultaneous execution of transactions is equivalent to a serial schedule [Bern 78]. At the instant when insurances are granted to one transaction, all resources utilized by this transaction are locked. Moreover, a successful transaction cannot release a lock before obtaining the insurances. Therefore, all successful transactions cannot lock a new resource after releasing a lock and subsequently reach a point (just after the insurances are granted) where all utilized resources are locked. It has been shown [Eswa 76, Gard 78] that any simultaneous execution of two phase transactions does not violate the consistency of any database. In fact, it is a necessary and sufficient condition to ensure consistency when we deal with an unknown set of transactions [Mino 78]. Since our algorithm forces all transactions to be two phase, using our algorithm for updating replicated databases ensures the consistency.

4. Recovery Procedure For A Down Site

To start up a repaired site after its failure, we must execute the following recovery procedure:

- 1) Reconstruct an old consistent version of

- the local database.
- 2) Find an operative site which has an up-to-date version of the database.
- 3) Update the local database with the modifications performed by this operative site which were not committed by the local database.
- 4) Reconstruct the lock tables, insurance status, and message queues for the repaired site.

These steps are described in detail in the following.

All modifications which are not recorded in the local journal are assumed to be lost when the site fails. The first step (reconstruction of a consistent database) can be accomplished by using differential files [Seve 76] and by deleting all the differential files which correspond to transactions whose lock tables are not recorded in the local journal. The reconstruction can also be done by using a classical recovery scheme with checkpoints and "before images" [Verh 78]. In this case, the checkpoints must be recorded in the lock table journal to cancel transactions which arrived after the checkpoints (for reconstructing the database).

The second step is to select an operative site which has performed all the updates on its own database copy. All sites which are up, that is those where a message can be sent without receiving a down condition, are possible candidates. If several sites are up, the criteria utilized for selecting the site is to minimize the cost of message transfers. The selected site is called the recovery site. Should this site fail during the recovery process, it is necessary to restart the recovery process with another up site.

The third step is to read the journal of the recovery site and to update the local copy of the database. The recovery procedure running at the recovery site checks to determine if each lock table is included in the local journal. If not, the procedure requests the corresponding updated records from the recovery site, then updates the records at the local database. It is sufficient to read only a part of the journal. The starting point of this part of the journal can be computed from the time stamp of the most recent lock table of the failed site minus the maximum amount of time needed for a complete execution of any transaction.

The last step copies the current status such as lock tables, insurance status, and message queues from the recovery site. This operation should be carried out instantaneously in order to get a consistent state. For this purpose, when the recovery procedure reaches the end of the journal at the recovery site, lock tables, insurance status and message queues of the current recovery site are sent together in one message with the last record of the journal. The failed site should be declared up before reading this

message and duplicated messages should be eliminated. Thus the recovery procedure can reconstruct the system status to be consistent with the local journal of the recovery site.

5. The Deadlock Detection Algorithm

5.1 The Waiting Queue

The denied lock requests are stored in a waiting queue at each local site. Whenever a transaction invokes the database system to perform an operation such as update, read, insert, delete, create, etc. on a set of records, the local database manager must lock the required resource(s). For this purpose, the LOCK procedure is executed. If the resource is not available, the request is stored in a waiting queue named WAIT. Each entry in this queue contains the transaction identifier and the address of four elements W, T, QID and QOD (see figure 2).

Figure 2. The Waiting Queue

For simplicity we shall designate the elements associated with transaction t in the waiting queue by $W(t)$, $T(t)$, $QID(t)$, $QOD(t)$. $W(t)$ is the identifier of the transaction which is waited for by transaction t , viz. the result from the execution of the LOCK function. $T(t)$ is the time transaction t has spent in the waiting queue. $QID(t)$ and $QOD(t)$ are the resource identifier and the operation mode vector, respectively, of the denied lock request.

5.2 Deadlock Detection Principles

Since wait conditions can occur only among transactions executed on the same site, a local deadlock detection procedure is sufficient. Such a procedure is initiated when a transaction t has exceeded a prescribed period of time; that is, when $T(t)$ is greater than a given value (for example 30 seconds). The principle of the deadlock detection algorithm is to check whether the transaction is involved in a cycle of transactions waiting for each other [Chu 74]. Since a transaction can wait for only one transaction, the detection is quite simple. Moreover, the algorithm determines the transaction which should be rolled back if a cycle is detected.

It is desirable to roll back a transaction of a user who has not waited for a long period of time and has not performed a large amount of computation. The roll back cost of a transaction is a trade off between these two quantities. We will compute this cost as a linear function of

the waiting time and the number of resources already locked by the transaction. The transaction with the lowest roll back cost among those involved in the cycle is chosen as the rolled back transaction.

5.3 Deadlock Detection Procedure

The deadlock detection algorithm is given in Procedure 5. The function DEADLOCK gives value 0 if transaction e is not involved in a deadlock and otherwise gives the identifier of the rolled back transaction. The procedure computes the roll back cost of a transaction (line 6), and selects the transaction with the minimum roll back cost (line 7). Next, we take the transaction being waited for as the new transaction (line 8). If this transaction is e , then there is a cycle and a deadlock is detected (line 9). If the transaction under consideration is not in the waiting queue, then there is no cycle and the transaction is not involved in a deadlock (line 5). Otherwise, the detection algorithm continues.

```

1. function DEADLOCK(e)
2. begin
3.   t := e ;
4.   cost := maximum possible value ;
5.   while t is in WAIT do
6.     c := a * T(t) + SIZE[RID (t,*)] ;
7.     if (c < cost) then cost := c; DEADLOCK := t; fi;
8.     t := W(t) ;
9.     if e = t then exit; fi ;
10.  od ;
11.  DEADLOCK := 0
12. end ;

```

Procedure 5: The Deadlock Detection Procedure

Note: In line 6, a is the coefficient of the cost function and SIZE gives the storage size of its parameter.

6. Performance Comparisons

6.1 Synchronization Algorithms for Comparison Studies

Several algorithms have been proposed to synchronize the updates of multiple copies of data bases. We will compare the proposed algorithm with the voting algorithm [Thom 78] and the centralized locking algorithm [Elli 77]. To compare algorithms under similar conditions, we use broadcasting as a message transfer scheme.

We shall first describe the voting algorithm for avoiding inconsistencies in updating replicated databases.

- 1) The algorithm uses a time stamp on each resource which reflects the time at which the resource was assigned its current value. Updates are accepted only when they are performed by transactions which have been started later than the current time stamp value of the resource.

- 2) When a transaction submits an update request, it includes a unique priority number, the identifier and the updated value of the resource to be modified, a list of the resources upon which the update is based (the base resources), and a list of time stamps corresponding to the base resource values used for the update.
- 3) When a site receives an update request, it must perform one of the following actions:
 - Vote "reject" if one or more of the base resource time stamps are older than those of the local site.
 - Vote "accept" only if the base resource time stamps are current and the request does not conflict with another active request (one which is neither accepted nor rejected). It conflicts if either an active request is trying to modify a base resource of the request under consideration, or the resource to be updated is a base resource for any other active request.
 - Vote "pass" if the base resources are current but the request conflicts with an active request that has a higher priority.
 - Defer voting if the base resources are current but the request conflicts with an active request of lower priority or if the base resource time stamps of the request are more current than the corresponding local resource time stamps.
- 4) When the requesting site has received a majority of accept votes and no reject votes for a request, then it performs the requested operation. It also notifies all other sites to perform the operation.

We now describe the centralized locking algorithm for avoiding inconsistencies:

- 1) Before performing an operation on a resource, a transaction requests the central controller (via its local system) to lock all copies of the resource.
- 2) The central controller checks the requested lock. If the request can be granted then a grant message is sent back to the site of the transaction. The transaction is then allowed to perform the update.
- 3) At the end of the transaction and after performing the two step commitment, the algorithm unlocks all resources utilized by the transaction.

6.2 Measures Used for Performance Comparison

Two important measures to compare performance of algorithms are: 1) Number of messages (NM) required to perform an update in a multiple copy database. 2) Volume of control messages (VM) required to perform an update in a multiple copy database. To simplify the comparison, the write

messages are not included in VM. We will use the above measures to compare the performance of the three algorithms with selected types of transactions.

Let N = number of sites in the distributed system,
 K = number of updates per transaction,
 c = number of control characters per message,
 r = size of a resource identifier,
 s = size of a time stamp,
 b = size of a command plus transaction identifier.

We assume that all the algorithms use a two step commitment protocol to ensure the reliability.

The voting algorithm requires $(N-1)/2$ vote request and $(N-1)/2$ grant messages per update*, $(N-1)$ write messages to send the 'new' value of the modified resource per update, and $3(N-1)$ messages for the commitment of a transaction (insure, grant, and done messages). Therefore, the number of messages required for a transaction with K updates is

$$(NM)_V = (N-1)(2K+3). \quad (1)$$

A vote request message requires transmission of the base variable identifiers with their time stamps, the modified resource identifier with its time stamp, a vote command, the transaction identifier and control characters. Let us assume that generally only one base variable is necessary. The size of a vote request message is $2(s+r)+b+c$ characters. The size of a grant message is the same as that of the commit message which is $c+b$ characters. Thus,

$$(VM)_V = (N-1)[(s+r+c+b)K+3(c+b)] \quad (2)$$

(in characters).

The centralized locking algorithm requires one lock request and one grant message per update, $(N-1)$ write messages per update, and $3(N-1)$ messages to commit a transaction. Therefore, the number of messages required for a transaction with K updates is

$$(NM)_C = (N-1)(3+K) + 2K. \quad (3)$$

A lock message requires the transmission of a lock command, a transaction identifier, a resource identifier and control characters. Therefore, the size of a lock message is $b+r+c$ characters. The size of a grant message or a commit message is $c+b$ characters. Thus, the volume of control messages is

$$(VM)_C = 3(N-1)(c+b) + K(r+2b+2c) \quad (4)$$

(in characters).

For the proposed algorithm, after local execution of a transaction, the algorithm transmits $(N-1)$ "lock table" and "grant" messages, $K(N-1)$ "write" messages, and $3(N-1)$ messages for the

commitment of a transaction. The number of messages* transmitted for a transaction with K updates is

$$(NM)_P = (N-1)(5+K). \quad (5)$$

The lock table size is $b+Kr+c$. The "grant," "insure," and "done" messages have the same size and are equal to $b+c$. Thus the volume of control messages transmitted per transaction is

$$(VM)_{P(\text{in characters})} = (N-1)(Kr+5b+5c) \quad (6)$$

6.3 Comparison of Performances

Let us first compare the number of messages required per transaction for the selected algorithms. Figures 3 and 4 present the number of messages as a function of the number of sites for a transaction with 5 updates and 20 updates, respectively. Figure 5 presents the number of messages as a function of the number of updates per transaction. From Figures 3, 4, and 5, we notice that the proposed algorithm yields better performance than that of the centralized algorithm.

Figure 3: Number of Messages vs. the Number of Sites with 5 Updates per Transaction

* Note that the number can be significantly reduced by packing several write messages together into a larger size message. This feature is inconvenient to implement in the voting and centralized algorithms because they usually perform the consistency verification at each update.

* We assume the request message is broadcasted only to a majority of sites.

Figure 4. Number of Messages vs. Number of Sites with 20 Updates per Transaction

Comparing Equations (1) and (5), we note that for the number of updates per transaction K greater than 2, the proposed algorithm yields a lower number of messages per transaction than the voting algorithm. Comparing Equations (1) and (3), we note that for a network consisting of more than three sites, the voting algorithm requires more control messages than the centralized locking algorithm. Finally, comparing Equations (3) and (5), we note that when the number of updates per transaction plus one ($K+1$) is less than the number of sites (N) in the network, the centralized algorithm yields fewer control messages per transaction than the proposed algorithm. Although the centralized algorithm is more efficient for large networks, it cannot always be adopted for implementation because of reliability and response time problems.

Let us now compare the volume of control messages transmitted with the following parameter values:

control characters per message, $c = 5$,
 resource identifier, $r = 5$,
 time stamp, $s = 5$,
 transaction identifier plus command, $b = 5$.
 From Equation (2), $(VM)^V = (N-1)(20K+30)$.
 From Equation (4), $(VM)^C = 30(N-1)+25K$.
 From Equation (6), $(VM)^P = (N-1)(5K+50)$.

Figure 6 presents the volume of control messages

Figure 5. Number of Messages vs. Number of Updates per Transaction

Figure 6. Volume of Control Information vs. Number of Sites

transmitted as a function of the number of sites for the selected algorithms. Each transaction is assumed to require 5 updates. The proposed algorithm yields performance comparable to the centralized algorithm for a network with few sites and yields better performance than that of the voting algorithm.

From the above comparison study, we note that the proposed algorithm performs better than the voting algorithm and yields performance comparable to that of the centralized algorithm (known as the most efficient algorithm [Garc 78]). Since the proposed algorithm uses distributed control, it is more reliable than the centralized locking algorithm.

7. Conclusion

We have proposed and proved the consistency of a distributed control, deadlock free algorithm for updating replicated databases. This algorithm utilizes both locking and time stamp schemes. To reduce overhead of the algorithm, locking is applied locally and time stamping is applied to the local lock table to detect conflicts among transactions executed at replicated database sites. As a result of this method of handling the lock tables and time stamps, the procedure for failure recovery is rather simple. Further, the storage overhead, the number of messages, and the volume of control messages required for a transaction is fairly low. Our performance study reveals that the proposed algorithm yields overhead lower than that of the voting algorithm and comparable to that of the centralized locking algorithm. Since the centralized algorithm has reliability as well as response time problems for a network with a large number of sites, the proposed algorithm should have a good potential for future use in distributed data bases.

Acknowledgments

The authors wish to express their thanks to Paul Hurley of UCLA and Jim Gray of IBM Research San Jose for their helpful comments and discussions.

References

- [Bern 78] Bernstein, P.A., J.B. Rothnie, N. Goodman, and C.A. Papadimitriou, "The Concurrency Control Mechanism of SSD-1: A System for Distributed Databases (The Fully Redundant Case)," IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, pp. 154-168, May 1978.
- [Coda 71] Codasyl Database Task Group ACM Report, New York, April 1971.
- [Chu 74] Chu, W.W., and G. Ohlmacher, "Avoiding Deadlock in Distributed Data Bases," Proceedings of the ACM National Symposium, Vol. 1, March 1974, pp. 156-160.
- [Elli 77] Ellis, C.A., "A Robust Algorithm for Updating Duplicate Databases," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 146-158.
- [Eswa 76] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," Comm. ACM, Vol. 19/11, Nov. 1976, pp. 624-633.
- [Garc 78] Garcia Molina, H., "Performance Comparison of Two Update Algorithms for Distributed Databases," Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978, pp. 108-122.
- [Gard 76] Gardarin, G. and S. Spaccapietra, "Integrity of Databases: a General Lock-out Algorithm with Deadlock Avoidance," in Modelling in DBMS, edited by J. Nijssen, North Holland Pub. Co., 1976, pp. 395-411.
- [Gard 77] Gardarin, G. and P. Lebeux, "Scheduling Algorithms for Avoiding Inconsistency in Large Databases," Third International Conference on Very Large Databases, Tokyo, October 1977.
- [Gard 79] Gardarin, G. and B. Piot, "Detection and Frequency Evaluation of Concurrency Conflicts," 1st International Conference on Distributed Processing, Toulouse, France, February 1979.
- [Lamp 76] Lampson, B. and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Internal Report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.
- [Lamp 78a] Lamport, L., "The Implementation of Reliable Distributed Multiprocess Systems," Computer Networks, Vol. 2, N2, May 1978.
- [Lamp 78b] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, Vol. 21, No. 7, pp. 558-564, July 1978.
- [Mena 78] Menasce, D.A. and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 28-31, 1978, pp. 215-234.
- [Mino 78] Minoura, T., "Maximally Concurrent Transaction Processing," Third Berkeley Workshop on Distributed Data Management and Computer Networks, August

1978, pp. 206-214.

- [Seve 76] Severance, D.G. and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," ACM TODS, Vol. 1/3, September 1976, pp. 256-267.
- [Ston 78] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Third Berkeley Workshop on Distributed Data Management and Computer Network, August 1978, pp. 235-258.
- [Thom 78] Thomas, R., "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," Proceedings of COMPCON 78, pp. 56-62, San Francisco, February 1978.
- [Verh 78] Verhofstad, J.S., "Recovery Techniques for Database Systems," Computing Surveys, Vol. 10, N2, June 1978.