DISTRIBUTED DATA BASES
C. Delobel and W. Litwin (eds.)
North-Holland Publishing Company

ENSURING CONSISTENCY IN A DISTRIBUTED DATABASE SYSTEM BY USE OF DISTRIBUTED SEMAPHORES*

Fred B. Schneider

Computer Science Department
Cornell University
Ithaca, New York 14853
U.S.A.

Solutions to the database consistency problem in distributed databases are developed. It is shown how any solution to the consistency problem for a centralized database system that involves locking can be adapted for use in distributed systems. This is done, constructively, in two steps. First, it is shown how locking can be implemented in terms of semaphores. Then, a semaphore implementation that is suitable for use in distributed systems is developed.

1. INTRODUCTION

A database can be viewed as a collection of entities, each of which has a value. The state of the database is defined in terms of the values of these entities. Typically, a database system will have a set of assertions, called consistency constraints, associated with it. These assertions characterize the valid states of the database. For example, in an accounting database one would expect to find the constraint "the sum of the debits and credits for each account is 0." If all the consistency constraints are satisfied by a database state D, then the database is said to be in a consistent state. This will be denoted C[D].

A user of a database system may view or alter portions of that database by means of transactions. A transaction is a sequence of primitive operations on the entities of the database. Some examples of primitive operations are: <read entity e_i>, <write entity e_j>. Execution of a transaction causes a mapping from one database state to another. Transactions are assumed to preserve consistency. Therefore:

(I): (∀ T: T a transaction: (∀ D: D a database state:
((C[D] and T(D) = D') ⟹ C[D'])))

A set of transactions is processed serially if transactions are processed one at a time, the next transaction being initiated only after completion of the previous one. Clearly, if a set of transactions is processed serially, and the database is initially in a consistent state, then the database state will be consistent immediately prior to, and after, each transaction is processed. This follows directly from (I), above. Notice that if a set of transactions is processed concurrently (i.e., not serially) the resultant state of the database may not be consistent. This is because transactions may "interfere" with each other; a transaction may reference some parts of the database as they were before execution of another transaction, and other parts as they were after that execution. The database consistency problem is concerned with devising and implementing schemes so that the result of concurrent execution of a set of transactions is the same as could be obtained by some serial execution of those transactions. Solution of this problem is important for the construction of database systems that support the concurrent

*This research was supported in part by NSF grant MCS-76-22360.

execution of a number of transactions. Typically, solutions involve defining restrictions on the execution of a transaction (e.g., "no entity may be locked after any entity has been unlocked"[3]).

In this paper, solutions for the database consistency problem in distributed databases are developed. It is shown how any solution for the consistency problem in centralized databases involving locks can be adapted for use in distributed systems. Distributed semaphores [10], an extension of a synchronization approach proposed by Lamport [4], are used for this purpose.

Section 2 of this paper discusses distributed systems, and some of the problems associated with implementing distributed database systems. Section 3 contains the development of some solutions to the database consistency problem in centralized database systems. These solutions are extended for use in distributed database systems in section 4 and section 5. Section 6 contains a discussion of our results and attempts to put them in perspective.

2. DISTRIBUTED DATABASE SYSTEMS

A distributed system is a system that is made up of more than one processor in which the only way processors can communicate with each other is by means of a communications network. Thus, in a distributed system processors do not directly have access to shared memory. Each of the processors, along with its memory and peripheral devices, is referred to as a site.

A distributed database system can be implemented by storing some subset of the entities that make up the database at each site s. Let e_s denote the set of entities stored at site s, and let E denote the set of all entities that make up the database. Any implementation where

$$E = \bigcup_{s \text{ a site}} e_s$$

is a distributed implementation of the database. There are many ways in which the entities that comprise E may be divided among the various sites. They can be characterized as follows. A distributed database system is fully redundant if every entity is stored at every site; partially redundant if some entities are stored at more than one site; and partitioned if no entity is stored at more than one site. Each organization has its advantages, depending on the nature of the database and its use. Note that in both fully redundant and partially redundant organizations some mechanism must be devised so that all copies of an entity have the same value. This is called the multiple copy consistency problem. We shall require any solution to the database consistency problem to solve the multiple copy consistency problem, as well.

For a number of reasons, the solution of the database consistency problem initially appears to be more difficult for distributed systems than for centralized systems (i.e., systems with shared memory). Solutions have been proposed ([1], [11] for example), but they tend to be difficult to understand and to verify formally. Reasoning about distributed systems is made difficult by the fact that no site can ever know the state of the entire system. The only way for sites to find out state information is by exchanging messages, which involve finite delays (of uncertain lengths). Thus, upon receiving such a message, the receiver can only learn of a past state of the originator of the message.

3. SOLUTION OF THE CONSISTENCY PROBLEM

Consider a database system where $T = \{t_i \mid 1 \le i \le n\}$ denotes the set of all possible transactions. The operation of the system in an environment in which there is one

(and only one) copy of each entity

DB: cobegin t

where cobegin S_1 || S_2 || ... || S
ments $S_1, S_2, \ldots, S_n$. If executio
the method of Owicki and Gries [6]
tion of DB can be obtained by seri
(This exercise is left to the read
This should not be surprising, sin
valid execution of DB must involve
(unspecified) serial order.

The atomicity of each of the trans
some of which have broader applica
action would appear to be atomic i
tive operations referred to one en
approach might require that no two
clearly restrict transactions unac
the atomic execution of each trans
restrictions.

A semaphore [2] is a non-negative
defined. The semantics of these o

{sem=c and c

{sem=c}

A single sempahore is used to guar
in the following program.

DB': var sm : semap

cobegin P(sm);

|| ... |

Again, it can be easily verified t
tency problem. Although the syste
execution of the transactions will
tions to the database consistency
of interleaving can be made much f
appear atomic with respect to each
that involve preemption, such as t
associate a lock bit with each ent
restricts when a transaction may l
fore, define restrictions on the s
protocols can be found in [3], [9]

Solutions that use lock bits can e
semaphores. Associate with each e

The primitive operations <lock ent
$P(sm_e)$ and $V(sm_e)$, respectively.
manner implement locking.

Consider any set of transactions T
solution to the database consisten

(and only one) copy of each entity can be modelled as follows.

DB: cobegin t_1 || t_2 || ... || t_n coend

where cobegin S_1 || S_2 || ... || S_n coend denotes concurrent execution of statements $S_1, S_2, \ldots, S_n$. If execution of each transaction is atomic, then by using the method of Owicki and Gries [6] it can be shown that the results of the execution of DB can be obtained by serial execution of the component transactions. (This exercise is left to the reader--it is tedious and not very enlightening.) This should not be surprising, since each transaction is atomic, and thus any valid execution of DB must involve executing the component transactions in some (unspecified) serial order.

The atomicity of each of the transactions could be ensured in a number of ways; some of which have broader applicability than others. For example, each transaction would appear to be atomic if it contained at most one primitive, and primitive operations referred to one entity and executed as atomic actions. A second approach might require that no two transactions reference the same entity. Both clearly restrict transactions unacceptably. By using a synchronization mechanism, the atomic execution of each transaction can be assured without unreasonable restrictions.

A semaphore [2] is a non-negative integer on which two operations, P and V, are defined. The semantics of these operations are as follows:

{sem=c and c>0} P(sem) {sem=c-1}

{sem=c} V(sem) {sem=c+1}

A single sempahore is used to guarantee the atomic execution of each transaction in the following program.

DB': var sm : semaphore initial (1);

cobegin P(sm); t_1; V(sm) || P(sm); t_2; V(sm);

|| ... || P(sm); t_n; V(sm) coend

Again, it can be easily verified that this is a solution to the database consistency problem. Although the system appears to be concurrent, in fact only serial execution of the transactions will result. However, by using one of the solutions to the database consistency problem that involves only locking, the grain of interleaving can be made much finer. These solutions ensure that transactions appear atomic with respect to each other. (Thus, we are precluding solutions that involve preemption, such as those found in [8].) Typically, these solutions associate a lock bit with each entity in the database and define a protocol that restricts when a transaction may lock and unlock entities. These solutions, therefore, define restrictions on the structure of transactions. Examples of such protocols can be found in [3], [9], and [7].

Solutions that use lock bits can easily be translated into solutions that use semaphores. Associate with each entity e a sempahore sm_e with initial value 1. The primitive operations <lock entity e> and <unlock entity e> are translated to $P(sm_e)$ and $V(sm_e)$, respectively. It should be clear that semaphores used in this manner implement locking.

Consider any set of transactions T that obey the restrictions of some "locking"

require that a transaction lock an entity prior to accessing it. A new set of transactions T' can be formed from those transactions in T, by translating the lock and unlock primitives into P and V operations as outlined above. The resultant system can be modelled by the following program:

DB": var sem_{e1}, sm_{e2}, ... sm_{em} : semaphore initial (1);

cobegin t'_1 || t'_2 || ... || t'_n coend

where the t'_i may contain P and V operations, and are atomic with respect to each other.

Solutions to the database consistency problem obtained in this manner have two drawbacks.

1. Semaphores have been defined in terms of shared memory. Thus, the use of semaphores will preclude the generalization of these solutions to distributed systems.

2. It was assumed that there was one (and only one) copy of each entity. This restricts the use of such solutions in fully or partially redundant database organizations.

These issues are addressed in the next two sections.

4. DISTRIBUTED SEMAPHORES

It is possible to implement semaphores without using shared memory. Semaphores implemented in such a fashion are called distributed semaphores [10]. A distributed semaphore is defined such that for every P operation that is completed by a process, an associated V operation has been performed.

In order to implement distributed semaphores, certain assumptions regarding the communications network are necessary. They are:

Broadcast Assumption: If a site broadcasts a message, that message will be received by every other site.

Message Order: All messages that originate at a given site are received by other sites in the order they were broadcast.

A timestamp ts(m) will be associated with each message m. It is assumed that timestamps are consistent with causality. Thus, if event V_1 can in any way affect V_2, the the timestamp associated with V_1 will be smaller than the one associated with V_2. Lamport has proposed a scheme to generate such timestamps without the use of a central clock or shared memory [4]. In that scheme, each site L has a unique integer name name(L) and a local clock c_L associated with it. Each local clock is updated as follows:

1. L increments c_L between any two successive events at L.

2. If event V is the broadcast of message m by site L, then m contains a timestamp ts(m) = c_L. Upon receipt of message m, site L' sets its clock $c_{L'}$ so that $c_{L'}$ = max[ts(m), $c_{L'}$+1].

Site names are used to resolve ties between identical timestamps, resulting in a total order of events that is consistent with causality.

For each distributed semaphore ds_i that is to be implemented, a message queue is

maintained at each site. This me
ascending order by timestamp--rec

Furthermore, it is assumed that w
knowledgement message is broadcas
are not acknowledged.) A message
ledgement message for m has been

Let V#(ds_i,x) be the number of "V
or equal to x that have been rece
ilarly for "P semaphore ds_i" mess
be computed from the message queu
and V operations on distributed s

V(ds_i): Broadcast message "

P(ds_i): Broadcast message "
timestamp on this m
m' concerning ds_i i

V#(ds_i

The derivation and proof of the c

It is not necessary to store the
site. Instead, the relevant info
a few integer variables. Due to
is fully acknowledged at site L,
ceived at L. Furthermore, the im
above requires only V#(ds_i,x) and

initial portion of the message qu
acknowledged message--can be stor
messages are received, they are p
that queue need not exceed the nu
m" becomes fully acknowledged, th
For every "P semaphore ds_i" messa

timestamp less than ts(m"), P# is
from the queue. The same is done

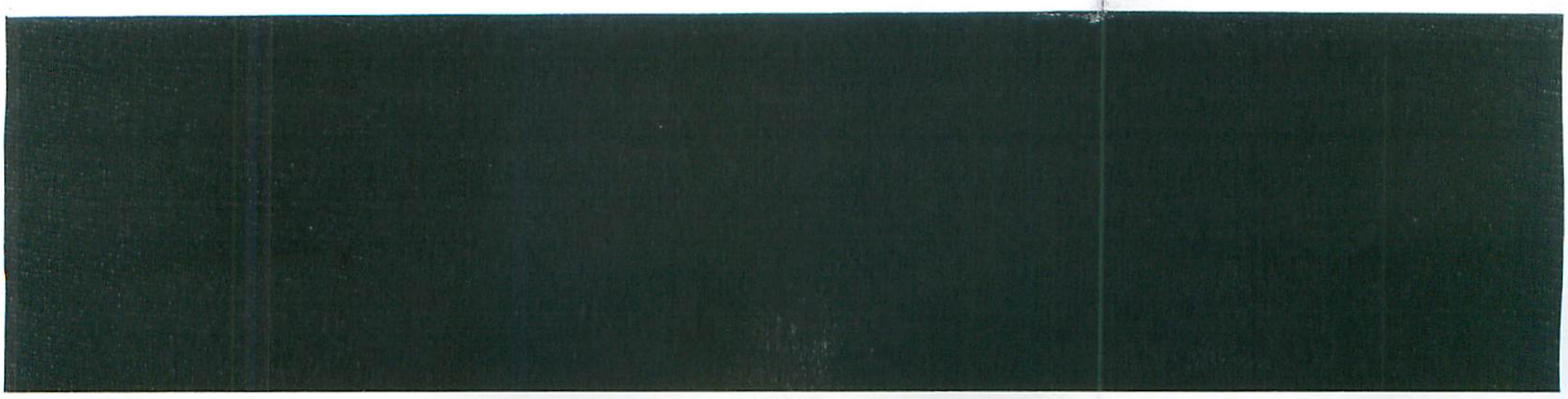
fore, in order to perform a P ope

site then waits until the value o
ed message queue is exceeded by t

By using distributed semaphores i
buted system any solution to the
Thus, DB" of section 3, in additi
in a centralized environment, wil
provided the issue of multiple co

5. THE MULTIPLE COPY CONSISTENCY

Consider the situation where ther
the database, as would be the cas
database. In order to satisfy th
copies of each entity must have t
Thus, in effect, the set of consi
tional assertions about the equal



maintained at each site. This message queue will contain messages--arranged in ascending order by timestamp--received by this site concerning semaphore ds_i.

Furthermore, it is assumed that whenever a message is received at a site, an acknowledgement message is broadcast to all other sites. (Acknowledgement messages are not acknowledged.) A message m is fully acknowledged at site L if an acknowledgement message for m has been received by L from every other site in the system.

Let $V\#(ds_i,x)$ be the number of "V semaphore ds_i" messages with timestamp less than or equal to x that have been received by the invoking site. Define $P\#(ds_i,x)$ similarly for "P semaphore ds_i" messages. The values of these functions can easily be computed from the message queue corresponding to semaphore ds_i at the site. P and V operations on distributed semaphores are then implemented as follows:

$V(ds_i)$: Broadcast message "V semaphore ds_i"

$P(ds_i)$: Broadcast message "P semaphore ds_i" and let tc denote the timestamp on this message. Then, wait until any message m' concerning ds_i is received and fully acknowledged and

$$V\#(ds_i,ts(m'))\geq P\#(ds_i,tc)$$

The derivation and proof of the correctness of this implementation appears in [10].

It is not necessary to store the entire message queue for each semaphore at every site. Instead, the relevant information from the message queue can be encoded in a few integer variables. Due to the message order assumption, after a message m is fully acknowledged at site L, no message m' where $ts(m') < ts(m)$ will be received at L. Furthermore, the implementation of distributed semaphores outlined above requires only $V\#(ds_i,x)$ and $P\#(ds_i,tc)$. Therefore, the information in the initial portion of the message queue--those messages up through the last fully acknowledged message--can be stored in two integer variables: P# and V#. As messages are received, they are put in a bounded message queue. The capacity of that queue need not exceed the number of sites in the system. Whenever a message m" becomes fully acknowledged, the values of P# and V# are updated as follows. For every "P semaphore ds_i" message currently in the bounded message queue with timestamp less than ts(m"), P# is incremented by 1. That message is then deleted from the queue. The same is done for V# and "V semaphore ds_i" messages. As before, in order to perform a P operation, a site broadcasts "P semaphore ds_i." The site then waits until the value of P# when that message is deleted from the bounded message queue is exceeded by the value of V#.

By using distributed semaphores it is therefore possible to implement in a distributed system any solution to the database consistency problem that uses locking. Thus, DB" of section 3, in addition to being a solution to the consistency problem in a centralized environment, will solve that problem in a distributed system, provided the issue of multiple copy entities is resolved.

5. THE MULTIPLE COPY CONSISTENCY PROBLEM

Consider the situation where there are multiple copies of some of the entities in the database, as would be the case in a partially or fully redundant distributed database. In order to satisfy the multiple copy consistency requirement, all copies of each entity must have the same value at the finish of every transaction. Thus, in effect, the set of consistency constraints has been augmented with additional assertions about the equality of all copies of a multiple copy entity.

Transactions preserve consistency, and therefore it may be necessary to alter the implementation of each transaction so that all copies of a multiple copy entity are updated. Note that a transaction need read only one copy of the entity (a local copy will suffice). This is because each entity is locked before it is accessed in a transaction. Consequently, at most one transaction may access a particular entity at any time. Furthermore, each transaction views the database in a consistent state--which now includes a requirement that all copies of an entity be equal. Therefore, at the beginning of the execution of a transaction, all copies of each of the entities referenced by the transaction are identical.

In order to update a multiple copy entity, a transaction broadcasts to all other sites a timestamped message containing the entity and its new value. Each site need save only those messages that deal with entities stored at that site. However, upon receipt of such a message, a site must broadcast an acknowledgement message to all other sites, even if the entity that is the subject of the message is not stored at that site. Note that the updates described by a message may not be applied to the database at site L until that message is fully acknowledged at site L. This is because prior to that time, other messages may be received that describe updates to the same entities, but have smaller timestamps. After that time, no such messages will be received (due to the message order assumption). The fact that update messages and distributed semaphore messages all use the same communications network (where message ordering holds over all messages) ensures that when a transaction executes, the local copy of every entity has its correct value. This is because, prior to accessing an entity, a P operation on a semaphore associated with that entity is performed resulting in the broadcast of a message that must be fully acknowledged for the P to complete. This serves to "flush" all update messages to that site from the communications network.

6. DISCUSSION

Distributed semaphores were originally developed to facilitate the solution of synchronization problems in distributed systems. We conjectured that the difference between a distributed synchronization problem and a centralized concurrent programming problem was merely one of implementation details, which could be ignored if a suitable synchronization mechanism was defined. This conjecture has been validated for the database consistency problem. It was shown how any solution to the database consistency problem that used locking could be adapted for use in distributed database systems.

Every attempt has been made here to make as few assumptions about the implementation environment as possible. Thus, the solutions developed are applicable in a broad range of systems. In general, optimizations require some detailed knowledge of the implementation environment; by making restrictions on the environment--knowing more of its properties--more optimization is possible. For example, if it is known that each site will initiate transactions with high frequency, then acknowledgement messages can be abolished. The acknowledgement by site L' for a message m that originated at site L is the receipt by L of message m' that originated at site L', where ts(m') > ts(m). Similarly, if a stipulation is made about the communications network--that it has "true" broadcast capability (as in Ethernet [5], for example) then to process a transaction in a system with n sites requires only n+2 transmission cycles (delays). This is derived as follows:

1	Broadcast "P" messages
n-1	Acknowledgement messages
1	Update non-local entities
1	Broadcast "V" messages
n+2	

This compares favorable with the bo

One of the often mentioned benefits signed so that they continue to fun The use of a particular synchroniza particular, note that in our implem fails, then no subsequent messages However, a more complex implementat that does not have these problems. a discussion of this mechanism.

ACKNOWLEDGEMENTS

G. Andrews, J. Archer, D. Gries, G. of this paper and furnished helpful

REFERENCES

[1] Bernstein, P.A., D.W. Shipman, Control Mechanism of SDD-1: A Corp. American, TR CCS-77-09,

[2] Dijkstra, E.W., Cooperating se ming Languages (Academic Press

[3] Eswaran, K.P., J.N. Gray, R.A. sistency and predicate locks i 624-633.

[4] Lamport, L., Time, clocks and CACM 21, 7 (July 1978) 558-565

[5] Metcalf, R.M., Ethernet: Distr works, CACM 19, 7 (July 1976)

[6] Owicki, S.S. and D. Gries, An I, Acta Informatica 6 (1976) 3

[7] Ries, D.R. and H. Stonebraker, management systems, ACM TODS 2

[8] Rosenkrantz, R.E. Stearns, and for distributed database syste

[9] Silberschatz, A. and Z. Kedem, to appear in JACM.

[10] Schneider, F.B., Synchronizati port 79-391, Computer Science

[11] Thomas, R.H., A solution to th data bases, Digest of Papers C

This compares favorable with the bounds obtained in [11].

One of the often mentioned benefits of distributed systems is that they can be designed so that they continue to function despite the failure of one or more sites. The use of a particular synchronization mechanism should not preclude this. In particular, note that in our implementation of distributed semaphores if a site fails, then no subsequent messages will ever be fully acknowledged at any site. However, a more complex implementation of distributed semaphores can be defined that does not have these problems. The interested reader is referred to [10] for a discussion of this mechanism.

ACKNOWLEDGEMENTS

G. Andrews, J. Archer, D. Gries, G. Levin and R. Schlichting read an earlier draft of this paper and furnished helpful comments.

REFERENCES

[1] Bernstein, P.A., D.W. Shipman, J.B. Rothnie, and N. Goodman, The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases, Computer Corp. American, TR CCS-77-09, December 1977.

[2] Dijkstra, E.W., Cooperating sequential processes, F. Genuys (ed.), Programming Languages (Academic Press, New York, 1968).

[3] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system, CACM 19, 11 (November 1978) 624-633.

[4] Lamport, L., Time, clocks and the ordering of events in a distributed system, CACM 21, 7 (July 1978) 558-565.

[5] Metcalf, R.M., Ethernet: Distributed packet switching for local computer networks, CACM 19, 7 (July 1976) 395-403.

[6] Owicki, S.S. and D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (1976) 319-340.

[7] Ries, D.R. and H. Stonebraker, Effects of locking granularity in database management systems, ACM TODS 2, 3 (September 1977) 233-246.

[8] Rosenkrantz, R.E. Stearns, and P.M. Lewis, System level concurrency control for distributed database systems, ACM TODS 3, 2 (June 1978) 178-198.

[9] Silberschatz, A. and Z. Kedem, Consistency in hierachical database systems, to appear in JACM.

[10] Schneider, F.B., Synchronization in a Distributed Environment, Technical Report 79-391, Computer Science Department, Cornell University (September 1979).

[11] Thomas, R.H., A solution to the concurrency control problem for multiple copy data bases, Digest of Papers COMPCON (1978).