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

By

Fred B. Schneider

TR79-392

Department of Computer Science
Cornell University
Ithaca, New York 14853

Ensuring Consistency in a Distributed Database System by Use of Distributed Semaphores*

9/11/79

Fred B. Schneider

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

Abstract

Solutions to the database consistency problem in distributed database 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.

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

1. Introduction

A database can be viewed as a collection of entities, each of which has a value. Thus, the state of the database is defined in terms of the values of the entities that comprise the database. 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. Thus:

(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 initially is 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 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" [E76]).

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 [S79], an extension of a synchronization approach proposed by Lamport [L78], 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 entitiy 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 ([B77], [T75] 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 orginator of the message.

3. Solution of the Consistency Problem

Consider a database system where $T = \{t_i \mid 1 \leq i \leq n\}$ 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 can be modelled as follows.

$$\text{DB: } \underline{\text{cobegin}}\ t_1 \parallel t_2 \parallel \ldots \parallel t_n\ \underline{\text{coend}}$$

where $\underline{\text{cobegin}}\ S_1 \parallel S_2 \parallel \ldots \parallel S_n\ \underline{\text{coend}}$ 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 [076] 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 [D68] 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 semaphore is used to guarantee the atomic execution of each transaction in the following program.

```
DB': var sm : semaphore initial (1);
     cobegin P(sm); t1; V(sm) || P(sm); t2; V(sm);
             || ... || P(sm); tn; 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 can result. The grain of interleaving can be made much finer by using one of the solutions to the database consistency problem that involves only locking. These solutions must ensure that transactions appear atomic with respect to each other. (Thus, we preclude solutions that involve preemption, such as those found in [R78].) 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 [E76], [S78] and [R77].

Solutions that use lock bits can easily be translated into solutions that use semaphores. Associate with each entity e a semaphore 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" solution to the database consistency problem. Note that any such solution requires 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 a distributed environment.

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 [S79]. 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, then 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 [L78]. 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 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 never acknowledged.) A message m is <u>fully acknowledged</u> 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 seamphore ds_i"

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

$$tc \leq ts(m') \quad \underline{\text{and}} \quad V\#(ds_i,ts(m')) \geq P\#(ds_i,tc)$$

The derivation and proof of the correctness of this implementation appears in [S79].

It is not necessary to store the entire message queue corresponding to each distributed semaphore at every site. The implementation outlined above requires only $V\#(ds_i,x)$ and $P\#(ds_i,x)$, where x is at least as large as the timestamp on the last message broadcast by this site. Furthermore, notice that 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. Therefore, the initial portion of the message queue--those messages up through the last fully acknowledged message--can be encoded in terms of two integer values, V# and P#. However, messages with timestamps greater than that of the last fully acknowledged message must be saved in a queue.

It is therefore possible, by using distributed semaphores, 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 database system, will solve that problem in a distributed environment, provided the issue of multiple copy entities is resolved.

5. The Multiple Copy Consistency Problem

Consider the situation where there is more than one copy of some 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 each copy of a multiple copy entity with the other copies. 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 will be identical.

In order to update a multiple copy entity, a transaction broadcasts to all other sites a timestamped message containing the name of the entity and its new value. Each site need save only those messages that deal with entities stored at that site. Upon receipt of such a message, a site broadcasts 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 in a message should 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

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 currect 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 developed to facilitate the solution of synchronization problems in distributed environments. The difference between a distributed synchronization problem and a centralized concurrent programming problem is merely one of implementation details, which can be ignored if a suitable synchronization mechanism is defined. By separating aspects of the problem dealing with the execution environment (such as the absence of shared memory) from those concerned with the logical properties of the solution (synchronization constraints, for example) the solution of the distributed synchronization problem is simplified. To il-

lustrate this, it was shown how any solution of the database consistency problem that used locking could be extended for use in distributed database systems.

The solutions developed here are not optimal in a number of respects. For example, many more messages are exchanged than is necessary. It is possible to eliminate acknowledgement messages and include that information in other messages. 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. Here every attempt has been made to make as few assumptions about the implementation environment as possible. Thus, the solutions developed are applicable in a broad range of systems. We favor the approach of developing simple, understandable solutions that are then optimized over the approach of constructing complex and ad hoc mechanisms that require complicated correctness proofs. (See [E77], for example.)

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 ack-

nowledged 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 [S79] for a discussion of this mechanism.

Acknowledgements

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

References

[B77] P.A. Bernstein, D.W. Shipman, J.B. Rothnie, and N. Goodman, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases," Computer Corp. America, TR CCA-77-09, Dec. 1977.

[D68] E.W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages, F. Genuys (Ed.), Academic Press, New York, 1968.

[E76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11 pp. 624-633.

[E77] C.A. Ellis, "Consistency and Correctness of Duplicate Database Systems," in Proc. of Sixth ACM Symposium on Operating System Principles (Nov. 1977), pp. 67-84.

[L78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," CACM 21, 7 pp. 558-565.

[O76] S.S. Owicki, and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica 6, pp. 319-340.

[R77] D.R. Ries, and H. Stonebraker, "Effects of Locking Granularity in Database Management Systems," ACM TODS 1, 2 pp. 233-246.

[R78] D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," ACM TODS 3, 2 pp. 178-198.

[S78] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database Systems," (to appear) in JACM.

[S79] F.B. Schneider, "Synchronization in a Distributed Environment," Technical Report, Computer Science Department, Cornell University.