DISTRIBUTED DATA BASES
C. Berlebach and W. Litwin (eds.)
North-Holland Publishing Company
© INRIA, 1980

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 constraints 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 e1>, <write entity e2>. Execution of a transaction causes a mapping from one database state to another. Transactions are assumed to preserve consistency. Therefore:

\[ (\forall T : T \text{ a transaction}: (\forall D: D \text{ a database state}: \quad (C[D] \text{ and } T[D] = D') \equiv C[D']) ) \]

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 (1) 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-76-22369.
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”[1]).

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 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

\[ E = \bigcup_{s \in S} E_s \]

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 about 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 \( T = \{ t_i \mid l_i \sigma_n \} \) 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.

\[ \text{DB:}\begin{align*}
\text{cobegin}\ s_1 \mid & \ s_2 \mid \ldots \mid s_n \\
\text{end} \end{align*} \]

where \( \text{cobegin} \ s_1 \mid s_2 \mid \ldots \mid s_n \text{end} \) means \( s_1, s_2, \ldots, s_n \). If execution of DB can be obtained by serial (This exercise is left to the reader. This should not be surprising, since valid execution of DB must involve (unspecified) serial order.)

The atomicity of each of the transactions some of which have higher applicability to be atomic operations referred to one another might require that the atomic execution of each trans restrictions.

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

\[ \text{(sem=c and e)} \]

\[ \text{(sem=e)} \]

A single semaphore is used to guard in the following program.

\[ \text{DB'}: \begin{align*}
\text{var sm : semap} \\
\text{cobegin P(sm);} \\
\text{end} \end{align*} \]

Again, it can be easily verified that this approach works. Although the system execution of the transactions will affect the database consistency of interleaving can be made much easier if various lock bits with each entity restricts when a transaction may begin, define restrictions on the lock protocols can be found in [3, 9].

Solutions that use lock bits can be considered. Associate with each entity. The primitive operations <lock i.e. \( P(sm_i) \) and \( V(sm_i) \), respectively. In a manner implement locking.

Consider any set of transactions \( T \) solution to the database consist...
ENSURING CONSISTENCY IN A DISTRIBUTED DATABASE SYSTEM

The problem in distributed data-...
and some of the problems proposed for use in distributed systems. Section 3 contains the problem in centralized systems. Section 3 contains the discussion of our re-

More than one processor in the system is by means of a method of two processors do not directly interlock with its memory and

Ensuring some subset of the entities do not the set of entities that make up the database.

There are many ways in which the database sites. They can be fully redundant if no copies are stored at more than one site. On the nature of the problem, we shall solve the multiple consistency problem.

The consistency problem is often for centralized systems proposed ([11], [11] and to verify formally, the fact that no site can find out the entity's delays (or under the receiver can only

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

\[ \text{DB: } \text{cobegin } t_1 || t_2 || \ldots || t_n \text{ coend} \]

where \text{cobegin } S_1 || S_2 || \ldots || S_n \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 [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 ensured 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:

\[
\begin{align*}
\text{sem} &= 0 \quad \text{P(sem)} \quad \text{sem} \\
\text{sem} &= 0 \quad \text{V(sem)} \quad \text{sem} + 1
\end{align*}
\]

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

\[ \text{DB': var sm: semaphore initial (1);} \]

\[ \text{cobegin } \text{P(sm); } t_1; \text{V(sm); } \text{P(sm); } t_2; \text{V(sm); } \]

\[ \ldots \text{P(sm); } t_n; \text{V(sm) coend} \]

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 a semaphore \( sm_e \) with initial value 1.

The primitive operations \langle lock entity e \rangle and \langle unlock entity e \rangle are translated to \( \text{P(sm)} \) and \( \text{V(sm)} \), 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" protocol. Note that a transaction cannot be preempted between a request to lock and the subsequent lock, and that no other transaction can lock an entity while a transaction is holding the lock.
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 result-
ant system can be modelled by the following program:

$$DB^*: \text{var } \text{sem} \ e_1', \ e_2', \ldots, \ e_m : \text{semaphore initial } (1) ;$$

$$\text{cobegin } t'_1 || t'_2 || \ldots || t'_n \text{ coend}$$

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 $V_2$, 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 [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 message queue is organized in ascending order by timestamp—recently received messages are at the head of the queue. Furthermore, it is assumed that when a knowledge message is broadcast, it is acknowledged. A message queue maintained at each site has been designed for this purpose. Let $V^+(ds_i,x)$ be the number of "V" operations that have been received at site $x$ for semaphore $ds_i$. Let $P(ds_i)$ be the broadcast message "P" for semaphore $ds_i$.

The derivation and proof of the correctness of this scheme is analogous to that of the broadcast of a message with no acknowledgment. By using distributed semaphores in a distributed system any solution to the database consistency problem obtained in this manner has been achieved. This approach is to be preferred in a centralized environment, where the size of the database is large and the overhead associated with broadcast of messages is high.

5. THE MULTIPLE COPY CONSISTENCY PROTOCOL

Consider the situation where there are multiple copies of each entity. This situation is encountered in a distributed database. In order to satisfy the consistency requirements for each entity, it must be possible to broadcast the message of a transaction to all copies of the entity. Thus, in effect, the set of consistency assertions about the equal
ENSURING CONSISTENCY IN A DISTRIBUTED DATABASE SYSTEM

maintained at each site. This message queue will contain messages—arranged in ascending order by timestamp—received by this site concerning semaphore $d_1$.

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 $V_i(d_{1i}, x)$ be the number of "V semaphore $d_{1i}$" messages with timestamp less than or equal to $x$ that have been received by the invoking site. Define $P_i(d_{1i}, x)$ similarly for "P semaphore $d_{1i}$" messages. The values of these functions can easily be computed from the message queue corresponding to semaphore $d_{1i}$ at the site. $P$ and $V$ operations on distributed semaphores are then implemented as follows:

$V(d_{1i})$: Broadcast message "V semaphore $d_{1i}$".

$P(d_{1i})$: Broadcast message "P semaphore $d_{1i}" and let $tc$ denote the timestamp on this message. Then, wait until any message $m$ concerning $d_{1i}$ is received and fully acknowledged and

$V(d_{1i}, ts(m)) = P_i(d_{1i}, tc)$

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 $V_i(d_{1i}, x)$ and $P_i(d_{1i}, x)$. 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: $P_i$ and $V_i$. 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 $P_i$ and $V_i$ are updated as follows. For every "P semaphore $d_{1i}" message currently in the bounded message queue with timestamp less than $ts(m)$, $P_i$ is increased by 1. That message is then deleted from the queue. The same is done for $V_i$ and "V semaphore $d_{1i}" messages. As before, in order to perform a $P$ operation, a site broadcasts "P semaphore $d_{1i}". The site then waits until the value of $P_i$ when that message is deleted from the bounded message queue is exceeded by the value of $V_i$.

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, $DB_n$ 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 acknowledgment 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 acknowledgment messages can be abolished. The acknowledgment 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 t(s(m')) > t(s(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 favorably with the box

One of the often mentioned benefits
signed so that they continue to fun
The use of a particular synchronization
particular, note that in our implemen
tails, then no subsequent messages
However, a more complex implementat
that does not have these problems.

ACKNOWLEDGEMENTS

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

REFERENCES


[2] Dijkstra, E.W., Cooperating se


[4] Lamport, L., Time, clocks and 


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


[10] Schneider, F.B., Synchronizat

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


