Network Working Group
RFC # 677
NIC # 31507

Paul R. Johnson (BBN-TENEX)
Robert H. Thomas (BBN-TENEX)
January 27, 1975

The Maintenance of Duplicate Databases

Preface:

This RFC is a working paper on the problem of maintaining duplicated databases in an ARPA-like network. It briefly discusses the general duplicate database problem, and then outlines in some detail a solution for a particular type of duplicate database. The concepts developed here were used in the design of the User Identification Database for the TIP user authentication and accounting system. We believe that these concepts are generally applicable to distributed database problems.

Introduction

There are a number of motivations for maintaining redundant, duplicate copies of databases in a distributed network environment. Two important motivations are:

- to increase reliability of data access.

  The accessibility of critical data can be increased by redundantly maintaining it. The database used for TIP login and accounting is redundantly distributed to achieve highly reliable access.

- to insure efficiency of data access.

  Data can be more quickly and efficiently accessed when it is "near" the accessing process. A copy of the TIP user ID database is maintained at each site supporting the TIP login service to insure rapid, efficient access. (Reliability considerations dictate that this database be redundantly maintained, and efficiency considerations dictate that a copy be maintained at each authentication site.)

The design of a system to maintain redundant, duplicate databases is a challenging task because of the inherent communication delay between copies of the database, as well as the real world constraints of system crashes, operator error, communication channel failure, etc. This paper discusses some of the problems we encountered in designing such a system, and outlines a system design for maintaining a particular type of database which solves those problems.

The Model

A system for supporting duplicate copies of a database can be modeled by a group of independent database management processes (DBMPs) each maintaining its own copy of the database. These processes communicate with each other over network communication paths. Each DBMP has complete control over its copy of the database. It handles all accesses to and modifications of the database in response to requests from other processes. Though the DBMPs act only upon requests, in the following they will often be said to be actually causing or originating the modifications.

An important design consideration is that the communication paths between the DBMPs are subject to failures. Thus one DBMP may have its interactions with other DBMPs interrupted and/or have to wait until communication paths are re-established before it can communicate with other DBMPs. An assumption made in this paper about the communication

paths is that messages from one process to another are delivered in the same order that they are sent. This is true of the ARPANET. For networks that make no such guarantee, communication protocols between the DBMPs can be used to correctly order the messages.

In order to proceed further, it is necessary to be more precise about the nature of the duplicated database and the operations allowed on it. A constant, read-only database is at one extreme. The task of the DBMPs is trivial in this case. They simply respond to value retrieval requests. At the other extreme is a general shared database where functional modification requests (such as "X := f(X,Y,Z)") are allowed and/or where it is necessary to completely restrict access to entries while they are being modified. In this case all the problems of shared databases on a single computer system arise (e.g., the need for synchronization mechanisms and the resulting potential deadlock situations), as well as those unique to having multiple database copies distributed among independent computers. For example, a completely general system must deal with the possibility of communication failures which cause the network to become partitioned into two or more sub-networks. Any solution which relies on locking an element of the database for synchronized modification must cope with the possibility of processes in non-communicating sub-networks attempting to lock the same element. Either they both must be allowed to do so (which violates the lock discipline), or they both must wait till the partition ceases (which may take arbitrarily long), or some form of centralized or hierarchical control must be used, with a resulting dependency of some DBMPs on others for all modifications and perhaps accesses as well.

The Database

The type of database to be examined in this paper can be represented as a collection of entries which are (Selector,Value) pairs. Each selector is unique and the values are atomic entities as far as the DBMPs are concerned. The mechanisms to be presented may be extended to handle databases with greater structure - where the values may themselves be collections of (selector,value) pairs - but this extension will not be considered further here.

Four operations are to be allowed on the database:

1) Selection - given a selector, the current associated value is returned.

2) Assignment - a selector and a value are given and the given value replaces the old value associated with the selector.

3) Creation - a new selector and an initial value are given and a new (selector,initial value) entry is added to the database.

4) Deletion- a selector is given and the existing (selector,value) entry is removed from the database.

Note that value modification is limited to assignment. Functional modification requests - such as "Change X to be Factorial(X)" - are specifically ruled out. Allowing them would force the use of system wide synchronization interlocks.

Consistency

The extent to which the copies of the database can be kept "identical" must be examined. Because of the inherent delay in communications between DBMPs, it is impossible to guarantee that the data bases are identical at all times. Rather, our goal is to guarantee that the copies are "consistent" with each other. By this we mean that given a cessation of update activity to any entry, and enough time for each DBMP to communicate with all other DBMPs, then the state of that entry (its existence and value) will be identical in all copies of the database.

Timestamps

We permit modifications to the database to originate at any of the DBMPs maintaining it. These changes must, of course, be communicated to the other DBMPs. To insure consistency, all of the DBMPs must make the same decision as to which modification to a particular entry is to be considered "final". It is desirable to select the "most recent" change. However, since there is no way to absolutely determine the time sequence of events in a distributed system without a universal, always accessible sequence number generator (a network time standard should suffice), "most recent" can only be approximated. We accomplish the approximation by associating a timestamp with each modification and with each entry, the latter being the timestamp of the modification which set its current value.(1) Since the uniqueness of timestamps given out at more than one

---

(1) Time is useful in this context because it has the desired properties of being monotonically increasing, and of being available with a reasonable degree of accuracy. Any other sequence numbering scheme with these properties can be used, "time-of-day" was chosen because it is simple to obtain. Its main faults are that it is often manually set (and thus prone to error), and it may stop during system service

interruptions. As computer systems learn to adapt to a network environment, the use of an independent time source should become more common. This is beginning to happen with the TENEX sites on the ARPANET.

DBMP can not be guaranteed, a "DBMP of origin" is included as part of each timestamp. By (arbitrarily) ordering the DBMPs, we thus have a means of uniquely ordering timestamps. Each
timestamp is a pair (T,D): T is a time, D is a DBMP identifier. For two timestamps (T1,D1) and (T2,D2) we have the following:

(T1,D1)>(T2,D2) <=> (T1>T2) or (T1=T2 and D1>D2)
(T1,D1) is said to be "more recent" than (T2,D2)

If D1=D2 and T1=T2 it is assumed that the two modifications are really two copies of the same modification request.

In order to insure the uniqueness of timestamps, it is necessary that each individual DBMP associate different times with different modifications. This is certainly possible to do, though the fineness of the unit of time may restrict the frequency of modifications at a single DBMP site.

Each entry in the data base is now a triple:
E ::= (S,V,T), where
S is the selector
V is the associated value
T is the timestamp (a Time,DBMP pair) of the last change to the entry

The task of each DBMP is to keep its copy of the database up-to-date, given the information on modifications that it has received so far. At the same time it must insure that each of its entries stays consistent with those of all the other DBMPs. This must be done despite the fact that the order in which it receives modifications may be very different from the order in which they are received by other DBMPs. In the remainder of this paper we examine the allowable database operations with respect to their effect on DBMP operation.

Assignment

Consider the case of assignment to an existing entry. When the assignment is initiated (by a person or process) the DBMP involved makes the change locally, and creates a copy of the modified entry and an associated list of DBMPs to which the change must be sent. As the modification is delivered to the other DBMPs, they are removed from the list until no DBMPs remain. The copy of the modification is then deleted. This distribution mechanism must be error free (i.e., receipt

of a modification must be positively confirmed before removing a DBMP from the list of recipients).(2)

When a DBMP receives an assignment modification (either locally or from another DBMP) it compares the timestamp of the modification with the timestamp of the copy of the entry in its database and keeps whichever is more "recent" as defined by the ordering given above. Thus when all existing assignments to a given entry have been distributed to all the DBMPs, they are guaranteed have the same "latest" value associated with that entry.

Creation

Creation and deletion of entries pose more of a problem. Note that the ability to create new, previously unknown entries requires that a DBMP be able to handle assignments to unknown entries. For example, consider the case of an entry XYZ created by DBMP A, and the following sequence of events: DBMP A tells DBMP B about the new entry, and subsequently B assigns a new value to XYZ; DBMP B then tells DBMP C about the assignment before C has heard from A about the creation. DBMP C must either save the assignment to XYZ until it hears about the creation, or simply assume the creation will be coming and use the "new" entry right away. The latter is more in the spirit of trying to keep the database as "up-to-date" as possible and leads to no inconsistencies.

Deletion

Deletion of entries is the main problem for this database system. If deletion is taken to mean immediate removal from the database, then problems arise. Consider the following scenario:

XYZ is an entry known by all DBMPs.
XYZ is deleted at DBMP A.
XYZ is modified at DBMP B (before B is notified of the deletion by A).

Now, consider a third DBMP, C. C may hear from DBMP B before DBMP A, in which case XYZ ends up deleted at DBMP C. C may however hear from DBMP A

---

(2) This same process (local modification and queuing for remote distribution) is, of course, performed for the other possible operations on the database. The details of how the local modification is done safely, how the messages are queued, how confirmation of delivery is done, etc., though important, will not be discussed here. The use of an addressee list attached to the modification to be delivered is conceptually easy to work with and not difficult to implement in practice.

before DBMP B. In this case, if C removes XYZ from its database, then the assignment to XYZ initiated by DBMP B will result in the re-creation of XYZ at DBMP C. To prevent this C must remember that XYZ has been deleted until it is "safe" to completely forget about XYZ.

One approach to this problem is to never actually remove an entry from the database. Deletion just marks the entry as being deleted by setting a "deleted" flag associated each entry. However, the problem of receiving assignments to deleted entries still exists. For example, DBMP A may receive an assignment from DBMP B to an entry which A has marked as deleted. DBMP A can not tell whether B has not heard about the deletion, or has heard about it but has also received a re-creation request for the entry, which hasn't reached DBMP A. To enable A to resolve such situations we include another timestamp in all entries: the timestamp of the entry's creation. Thus in the above example, DBMP A can compare the creation timestamps of the assignment and the deleted entry. The one with the later creation timestamp is kept. Indeed whenever a modification with an old creation timestamp is received it can be ignored.

We now have a 5-tuple for entries and modifications:

E ::= (S,V,F,CT,T)
S is the selector
V is the associated value
F is the deleted/not-deleted flag
CT is the timestamp of creation
T is the timestamp of this (last) modification

Note that the values of the F, CT, and T components of a modification uniquely specify the type of modification. Thus only the 5-tuple to become the new entry for a selector, not the type of modification, need be communicated:

F = not deleted, CT = T => creation
F = not deleted, CT < T => assignment
F = deleted => deletion

The mechanism described above handles all the desired operations on the distributed database in a way that guarantees the consistency of all copies. A modification to the database will take effect at each DBMP as soon as it receives the request from the DBMP originating the change.

A deficiency with this scheme is that deleted entries are never removed from the database. A method which permits "garbage collection" of deleted entries is discussed below.

Removal of Deleted Entries

The basic constraint is that a DBMP should not remove a deleted entry until it will never receive any assignments with the same selector (S) and the same or older create time (CT). If it fails to do this, then it will be unable to distinguish these "out of date" assignments from assignments to a newly created entry for the same S. To be able do this, each DBMP must know for each deleted entry whether all other DBMPs have heard about the deletion. To accomplish this, each DBMP could notify the other DBMPs whenever it hears about a deletion. If these notifications are transmitted in order with the "normal" sequence of modifications, then upon receipt of such a notification a DBMP can be sure that the sending DBMP has delivered any outstanding assignments to the deleted entry, has marked it as deleted, and will not generate any new assignments to it. Keeping track of, and exchanging messages about, each individual deleted entry in this manner is, however, somewhat more elaborate than necessary.

Having each DBMP deliver all its own modifications in sequential order (by timestamp) allows the following simplification. We have all DBMPs maintain a table of the timestamps of the last modification received from each other DBMP. Any DBMP, say A, is then guaranteed to have received all modifications originating at another DBMP, say B. which have timestamps earlier than (or equal to) the entry for B in A s copy of this table. If this table is exchanged between DBMPs, then all DBMPs would have a second N*N (N= number of DBMPs) table where entry (I,J) would be the timestamp of the last modification received by DBMP I from DBMP J. Thus DBMP A can remove a deleted entry whose deletion originated at DBMP K when all entries in the Kth column of this table at DBMP A are later than or equal to the timestamp of the deleted entry. When a DBMP receives a deletion modification, in addition to acting on it and acknowledging receipt of it, the DBMP should also send its table of last timestamps received to all other DBMPs. This is sent in a timestamped message which is queued with the "normal" modification messages.

A refinement to this approach, which reduces the amount of information exchanged and the size of the tables, is to have the DBMPs exchange only the oldest of the entries in the first table (of timestamps of last modifications received from other DBMPs). These would then be saved in a 1*N table, replacing the N*N table. A DBMP receiving a modification with a timestamp equal to or older than the oldest timestamp in its second table knows that the modification has been confirmed as being received by all other DBMPs. A deleted entry can thus be removed when its timestamp satisfies this condition. A DBMP would, upon receipt of a deletion modification, queue up a message with this "timestamp of oldest last modification received" for delivery to all other DBMPs.

Summary of solution:

An entry in the database is a 5-tuple:
(S,V,F,CT,T) where
S is an selector used in all references to this entry.
V is its present value.
F is a deleted/undeleted flag.
CT is the timestamp of the creation of this entry.
T is the timestamp of the modification which set the current V and/or F of the entry.
A timestamp is a pair (time,DBMP) where the DBMP identifies the site at which the time was generated, and the DBMPs are (arbitrarily) ordered, so that timestamps are completely ordered.

A modification is a pair (Set-of-DBMPs,Entry) where Set-of-DBMPs is the set of DBMPs to which the Entry has yet to be delivered.

An ordered (by timestamp) list of modifications is kept at each DBMP. The DBMP periodically attempts to deliver modification requests to those DBMPs which remain in the Set-of-DBMPs associated with each modification. Modification entries are removed from this list when they have been delivered to all DBMPs.

When a DBMP receives a modification request from another DBMP, it compares the timestamps of the request with the timestamps of the corresponding entry (if any) in its database, and acts upon or disregards the new entry accordingly.

Each DBMP keeps a vector of the timestamp (T) of the last modification received by it from each other DBMP.

When a DBMP receives a deletion modification, it sends a timestamped message to all other DBMPs containing the oldest timestamp in its vector of timestamps of last modification received. Each DBMP keeps a second vector of the last of these timestamps received from each other DBMP.

A deleted entry may be removed from the database when its timestamp (T) is older than all the timestamps in this second vector of timestamps received from other DBMPs.

Conclusion

This paper has presented techniques by which a number of loosely coupled processes can maintain duplicate copies of a database, despite the unreliability of their only means of communication. The copies of the database can be kept "consistent'. However it is possible for seemingly anomalous behavior to occur. For example a user may assign a value to a selector using one DBMP, later use another DBMP and assign a new value, and still later find that the first value is the one that remains in the database. This can occur if the clocks used by the two DBMPs for their timestamps are sufficiently out of synchrony that the second assignment is timestamped as having taken place before the first assignment. To the extent that the communication paths can be made reliable, and the clocks used by the processes kept close to synchrony, the probability of seemingly strange behavior can be made very small. However, the distributed nature of the system dictates that this probability can never be zero.

The major innovation presented here is the explicit representation of the time sequence of modifications through timestamps for both modifications and entries. This enables the each DBMP to select the same "most-recent" modification of an entry. In addition, timestamps enable the DBMPs to decide when a deleted entry is no longer referenced (i.e., still active anywhere) and can be deallocated. These techniques should have broader utility in building and modeling other systems of concurrent, cooperating processes.