1983

A comparative study of concurrency control algorithms for distributed databases

Fabio Aparicio

Rochester Institute of Technology
School of Computer Science and Technology

A Comparative Study of Concurrency Control Algorithms for Distributed Databases

by

Fabio Aparicio

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by: _____

Professor Rayno D. Niemi

_____

Professor John L. Ellis

_____

Professor Guy Johnson

December 2, 1983

I hereby grant permission to the Wallace Memorial Library, of Rochester Institute of Technology, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

By: _____

Date: Dec 15, 1983

ABSTRACT

The declining cost of computer hardware and the increasing data processing needs of geographically dispersed organizations have led to substantial interest in distributed data management. These characteristics have led to reconsider the design of centralized databases. Distributed databases have appeared as a result of those considerations.

A number of advantages result from having duplicate copies of data in a distributed databases. Some of these advantages are: increased data accesibility, more responsive data access, higher reliability, and load sharing.

These and other benefits must be balanced against the additional cost and complexity introduced in doing so. This thesis considers the problem of concurrency control of multiple copy databases. Several synchronization techniques are mentioned and a few algorithms for concurrency control are evaluated and compared.

KEYWORDS: distributed database management systems, transactions, concurrency control techniques, concu - rrency control mechanisms, and distributed programming.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Seldom does the generation of information concentrate on a unique source. Many institutions, such as governments, banks, airlines and chain stores are facing the problem of having their valuable information stored in computers which are geographically dispersed and either connected via communication links or not connected at all. This characteristic has led to the modification of the conventional designs of centralized architectures which then become distributed architectures. Distributed architectures overcome the problem outlined earlier and increase the reliability, accessibility, responsiveness and load sharing in the system. Thus studies to facilitate the understanding of this technology are of great help in the implementation of such systems.

The overall goal of the thesis is to study in detail some aspects related to concurrency in distributed systems and at the same time facilitate its understanding and development.

Chapter 2 illustrates, by means of two typical examples, the architectures of a distributed database and the

sort of transactions which exist in a distributed environment. Basic concepts like consistency, transaction, replication and transparency are also included.

Chapter 3 defines and illustrates some of the situations, or problems, found in the management of concurrent transactions in distributed databases. At the same time more basic concepts and terminology in this field are introduced. Once these problems have been stated, the corresponding characteristics of the concurrency control mechanisms are determined. These characteristics are classified as correctness, efficiency, reliability and generality.

Chapter 4 includes an introduction to concurrency control and its synchronization techniques used in distributed databases. It also includes a brief description of some recently proposed algorithms for concurrency control in distributed databases.

Chapter 5 includes a summary of previous work related to the evaluation of synchronization mechanisms, and sets forth criteria used to compare these mechanisms. A qualitative and quantitative evaluation of some of the previously described algorithms is included.

Chapter 6 includes an introduction on distributed programming. It also includes the code of three of the algorithms studied in previous chapters. Due to the lack of a well-known distributed programming language, psuedocode is used to code these algorithms. These algorithms are compared and conclusions are drawn.

CHAPTER 2

DESCRIPTION OF TWO TYPICAL EXAMPLES OF DISTRIBUTED DATABASES

2.1. INTRODUCTION

This chapter illustrates, by means of two examples, the architectures of a distributed database (DDB) and the sort of transactions which may exist in a DDB.

2.2. BASIC CONCEPTS

A database, in general, may be seen as a set of data and a database management system (DBMS). The DBMS is formed by modules that handle, among other functions, the user interaction, consistency control, reliability, concurrency, and data protection.

A distributed database has more elements in its architecture. There are data and DBMSs (these may be different) in various machines (nodes) which should be able to cooperate. Each computer has a processing capability and data storage capacities. Furthermore, each node may implement whatever data management and transaction management system it wants to. The various computers are connected by

communications links, and it is normal for these links to operate at a relatively low speed.

The data, dispersed on various machines, may be totally replicated, partially replicated, or partitioned (no replication at all . A number of advantages can result from keeping duplicate databases. Some of these advantages are [THOM79]:

(1) Increased data accessibility as the data may be accessed even when some of the nodes where it is stored have failed, as long as at least one of the sites is operational,

(2 More responsive data access; database queries initiated at nodes where the data are stored can be satisfied directly, and

(3) Load sharing as the computational load of responding to queries can be distributed among a number of database sites rather than centralized at a single site.

If all the data reside at the same node, the system is called centralized.

User interaction with the database is done through application programs (containing transactions), which communicate with the DBMSs. A transaction is a unit of work

which takes the database from a consistent state to another consistent state [BORR81,DATE83]. Transactions are formed by sequences of atomic actions. Transactions are an all or nothing thing, either they happen completely or all trace of them (except in the log) is erased [GRAY78]. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is undone thereby returning the data base to a consistent state. Thus transactions are also the units of recovery [GRAY76]. A transaction should have ways to indicate the beginning of it and the successful or unsuccessful termination of it.

An example of a typical trarsaction is [GRAY78] :

```
DEBIT_CREDIT :
     BEGIN_TRANSACTION ;
     GET MESSAGE ;
     EXTRACT ACCOUNT_NUMBER,DELTA,TELLER,
             BRANCH FROM MESSAGE ;
     FIND ACCOUNT (ACCOUNT_NUMBER) IN DATABASE :
     IF NOT FOUND | ACCOUNT_BALANCE + DELTA < 0
     THEN  PUT NEGATIVE RESPONSE ;
     ELSE  DO ;
           ACCOUNT_BALANCE = ACCOUNT_BALANCE + DELTA ;
           CASH_DRAWER(TELLER) = CASH_DRAWER(TELLER)+DELTA ;
           BRANCH_BALANCE(BRANCH) = BRANCH_BALANCE(BRANCH)
                                    + DELTA ;
           PUT MESSAGE ('NEW BALANCE =' ACCOUNT_BALANCE) ;
     END ;
     COMMIT ;
```

Transactions can be classified as follows :

- Simple : takes in a single message, does something, and

then produces a single message.

- Conversational : sends and receives several asynchroncus messages, and is likely to last several minutes (while the user thinks and types) and hence poses special resource management problems. Conversational transactions carry on a dialogue with the user.

- Batch : in general is not on-line and usually performs thousands of data management calls before terminating.

- Distributed : accesses data or terminals at several nodes of a computer network. It is a transaction but has instances (called transaction incarnations in [MENA79]) in several nodes. All the instances cooperate in the execution of the transaction.

- Read-Only : reads data from the database and has no write actions. Read-Only transactions are also known as Queries.

- Local : does work in the node where it was generated. Local read-only transactions are called Insular Queries in [GARC82a].

- Update : updates the contents of the database.

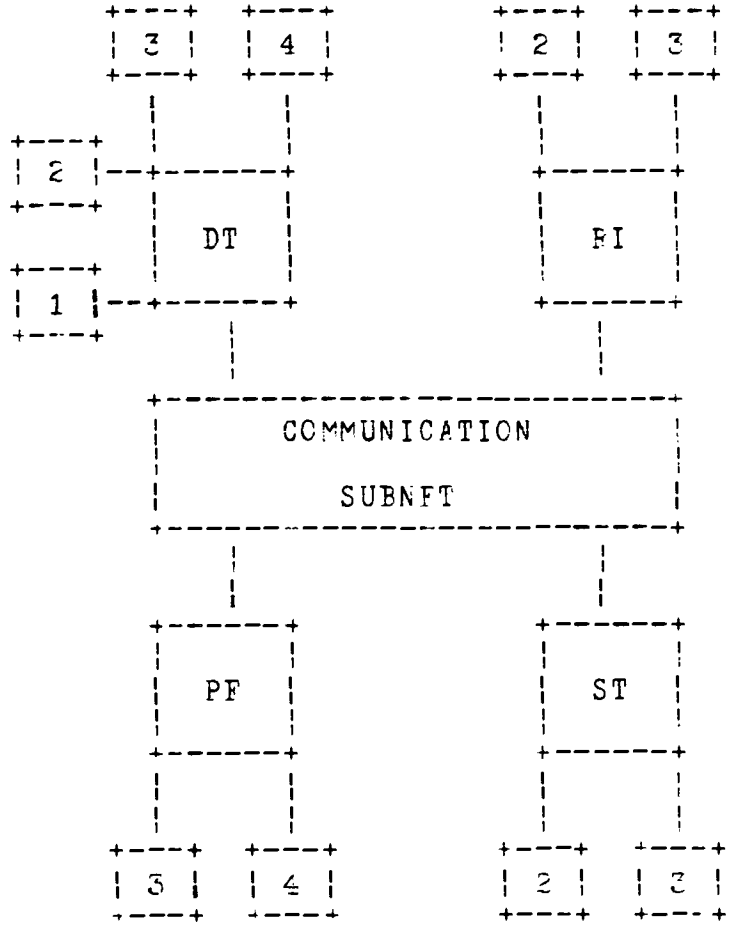
2.3. BANK DATABASE

A bank is a financial organization, which due to the nature of its business, is bound to have a DDB to insure its customers receive good service and to reduce its operating costs. 'BANK OF THE CITY', a hypothetical bank, includes a main office downtown and subsiciary branches spread throughout the suturbs. The bank has a DDB which is comprised of the files of its branches. Each node on the network is a branch bank (Figure 2.1).

The database is distributed as follows :

- Main Office Downtown : contains files of checking accounts opened at Downtown, R.I.T. , Southtown , and Pittsford.
- R.I.T. Branch : contains files of checking accounts opened at R.I.T. and Southtown.
- Southtown Branch : contains files of checking accounts opened at Southtown and R.I.T.
- Pittsford Branch : contains files of checking accounts opened at Pittsford and Southtown.

BANK DATABASE (Partial Peplication)



1. Checking Accounts Downtown (DT)
2. Checking Accounts R.I.T. (RI)
3. Checking Accounts Southtown (ST)
4. Checking Accounts Pittsford (PF

Fig. 2.1 Bank Database (Partial Replication)

Each node contains the above information, as well as the total assets from the branches on which it maintains files. This distribution is partially replicated and is shown in figures 2.2 , 2.3 , 2.4 , and 2.5.

Main Office Downtown

ACCOUNTS

BRANCH	ACCOUNT	BALANCE
Downtown	1-15-DT	5000
Downtown	1-16-DT	6000
Downtown	1-17-DT	3020
R.I.T.	2-15-RI	3000
R.I.T.	2-16-RI	2000
R.I.T.	2-17-RI	1000
Southtown	3-15-ST	3500
Southtown	3-16-ST	2500
Southtown	3-17-ST	3000
Pittsford	4-15-PF	10000
Pittsford	4-16-PF	1000
Pittsford	4-17-PF	1000

ASSETS

BRANCH	TOTAL
Downtown	14020
R.I.T.	6000
Southtown	9000
Pittsford	12000

Figure 2.2 Data Base Downtown.

PITTSFORD BRANCH

ACCOUNTS

BRANCH	ACCOUNT	BALANCE
Southtown	3-15-ST	3500
Southtown	3-16-ST	2500
Southtown	3-17-ST	3000
Pittsford	4-15-PF	10000
Pittsford	4-16-PF	1000
Pittsford	4-17-PF	1000

ASSETS

BRANCH	TOTAL
Southtown	9000
Pittsford	12000

Figure 2.3 Data Base Pittsford.

The total of each branch implies the first consistency constraint in the database; The sum of a branch account's balances is equal to the sum of the branch's assets. Other constraints are : the balance of an account must not be

R.I.T. BRANCH

ACCOUNTS

BRANCH	ACCOUNT	BALANCE
R.I.T.	2-15-RI	3000
R.I.T.	2-16-RI	2000
R.I.T.	2-17-RI	1000
Southtown	3-15-ST	3500
Southtown	3-16-ST	2500
Southtown	3-17-ST	3000

ASSETS

BRANCH	TOTAL
R.I.T.	6000
Southtown	9000

Figure 2.4 Data Base R.I.T.

SOUTHTOWN BRANCH

ACCOUNTS

BRANCH	ACCOUNT	BALANCE
R.I.T.	2-15-RI	3000
R.I.T.	2-16-RI	2000
R.I.T.	2-17-RI	1000
Southtown	3-15-ST	3500
Southtown	3-16-ST	2500
Southtown	3-17-ST	3000

ASSETS

BRANCH	TOTAL
R.I.T.	6000
Southtown	9000

Figure 2.5 Data Base Southtown.

negative , and two accounts can not have the same identification number.

We say that a DDB is consistent (or is in a consistent state if all the consistency constraints are satisfied by the data values [GARC82a]. This control is done automatically by the consistency subsystem of the DBMS.

Under this architecture. a user may interact with the data base by means of different classes of transactions.

LOCAL TRANSACTIONS

Local transactions refer to transactions which work on data existent on the node where they were gererated.

Example :

At the Southtcwr branch, a user whose account is 3-15-ST wants to know his balance.

Call_transaction : INQ_BALANCE (3-15-ST)

```
TRANSACTION : INQ_BALANCE (ACCT#
        FIND ACCOUNT (ACCT#) IN DATABASE ;
        IF NOT FOUND
        THEN PUT NEGATIVE RESPONSE ;
        ELSE DO ;
             PUT MESSAGE ('BALANCE = 'ACCOUNT_BALANCE);
        COMMIT ;
```

DISTRIBUTED TRANSACTIONS

Distributed transactions refer to transactions which involve data stored in various nodes.

Following the above example the user depcsits $ 1000.

Call_transaction : DEPOSIT (3-15-ST , 1000

```
TRANSACTION : DEPOSIT (ACCT# , VALUE)
        FIND ACCOUNT (ACCT#  IN DATABASE ;
        IF NOT FOUND
        THEN PUT NEGATIVE RESPONSE ;
        ELSE DO ;
             ACCOUNT_BALANCE = ACCOUNT_BALANCE
                               + VALUE ;
             PUT MESSAGE ('BALANCE =' ACCOUNT_BALANCE  ;
        COMMIT ;
```

This transaction is distributed because it has to update the files of the Downtown , R.I.T. , and Southtown branches in order to preserve the consistency of the database.

2.4. INVENTORY DATABASE

A chain store with warehouses in several localities (cities, counties ,etc.) is another example used to illustrate a DDB. The ACME Company sells furniture, and stores its goods in warehouses in three different cities : Another typical transactions are :

```
Call_transaction : OPEN ( R.I.T. , 2-20-PI , 3500 )

    TRANSACTION : OPEN ( BRANCH , ACCT# , BALANCE
            FIND ACCOUNT ( ACCT# ) IN DATABASE ;
            IF FOUND
            THEN PUT NEGATIVE RESPONSE ;
            ELSE DO ;
                 WRITE ACCOUNT ( BRANCH,ACCT#,BALANCE) ;
            COMMIT ;


Call_transaction : FUND_TRANSF ( 2-15-PI,3-17-ST,500 )

TRANSACTION : FUND_TRANSF ( ACCT1,ACCT2,VALUE
        FIND ACCOUNT (ACCT1) IN DATABASE ;
        IF NOT FOUND
        THEN PUT NEGATIVE RESPONSE ;
        ELSE DO ;
        FIND ACCOUNT (ACCT2) IN DATABASE ;
        IF NOT FOUND
        THEN PUT NEGATIVE RESPONSE ;
        ELSE DO ;
        ACCOUNT_BALANCE (ACCT1  = ACCOUNT_BALANCE (ACCT1
                                  - VALUE ;
        ACCOUNT_BALANCE (ACCT2) = ACCOUNT_BALANCE (ACCT2)
                                  + VALUE ;
        UPDATE ACCOUNT ( ACCT1 ) ;
        UPDATE ACCOUNT ( ACCT2 ) ;
        COMMIT ;
```

Rochester, Buffalo ; and Syracuse. Each node of the network is represented by a city (Figure 2.6).

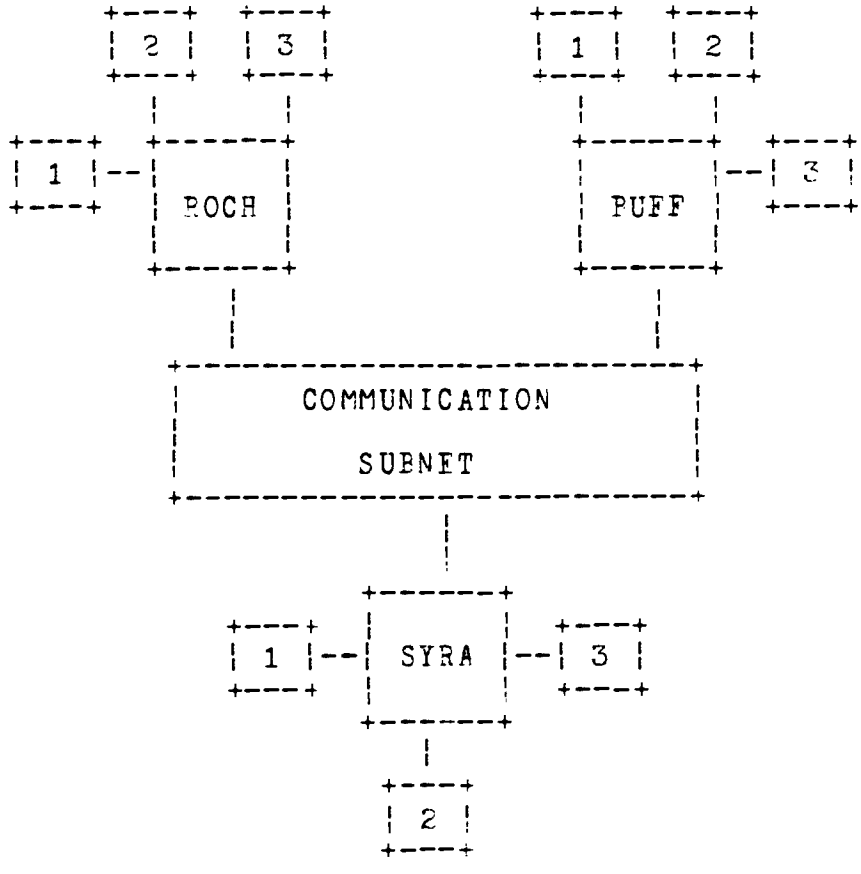
The database has the following distribution for each node :

- Rochester Node : articles stored in warehouses in Rochester, Buffalo, and Syracuse.
- Buffalo Node : articles stored in warehouses in Rochester, Buffalo, and Syracuse.
- Syracuse Node : articles stored in warehouses in Rochester, Buffalo, and Syracuse.

This kind of distribution, where every node has a copy of the whole database, is called total replication.

Each node morecver, maintains the total distribution of articles for every line of furniture as is shown in figure 2.7.

INVENTORY DATABASE (Total Replication)



1. Articles stored in Rochester.
2. Articles stored in Buffalo.
3. Articles stored in Syracuse.

Figure 2.6 Inventory Database.

Warehouses : ROCHESTER
BUFFALO
SYRACUSE

Database

CITY	ARTICLE	AMOUNT	UN.VALUE
ROCH	CHAIRS	10000	50
ROCH	SOFAS	3000	100
BUFF	SOFAS	2000	125
BUFF	DESKS	1234	150
SYRA	CHAIRS	3216	55
SYRA	DESKS	236	150

ARTICLE	TOTAL
CHAIRS	13216
SOFAS	5000
DESKS	1470

Figure 2.7 Warehouses.

Some typical transactions in this application are :

```
INQUIRY :
        TRANSACTION : INQ_AMOUNT ;
             READ AMOUNT OF CHAIRS IN ROCHESTER ;
             READ UNIT VALUE OF SOFAS IN SYRACUSE ;
        END TRANSACTION ;

UPDATE :
        TRANSACTION : UPD_AMOUNT ;
             READ AMOUNT OF SOFAS IN SYRACUSE ;
             AMOUNT = AMOUNT - 100 ;
        END TRANSACTION ;


TRANSFER :
        TRANSACTION : TRANSF_ARTICLES ;
             READ AMOUNT OF CHAIRS IN ROCHESTER ;
             AMOUNT = AMOUNT - 85 ;
             READ AMOUNT OF CHAIRS IN BUFFALO ;
             AMOUNT = AMOUNT + 85 ;
        END TRANSACTION ;
```

```
SET UPDATE :
        TRANSACTION : UPD_PRICES ;
             INCREASE BY 10% UNIT VALUE OF CHAIRS
             AT ALL LOCATIONS ;
        END TRANSACTION ;
```

With these examples of banks and inventories, we tried to show the external form of a DDB in two specific applications. By means of the typical transactions shown, we showed how the location and replication of data, as well as the necessary messages to completely execute a transaction, are transparent to the user.

Transactions should provide the programmer with the following types of transparencies :

- Location Transparency. Location transparency means that users and user programs should not need to know the site location of any particular data item [DATA83].

- Replication Transparency. Replication transparency means that all details of locating data and maintaining replicas up-to-date should be handled by the system, not by the user [DATE83].

- Concurrency Transparency. Concurrency transparency means that a user has the illusion that it is executing alone. The system must protect each user from the others.

- Failure Transparency. Failure transparency means that the

system should free users from taking precautions against failures.

Together, location transparency and replication transparency imply that (ideally) a distributed system should look like a centralized system to the user [DATE83].

CHAPTER 3

CONCURRENCY IN DISTRIBUTED DATABASES

3.1. INTRODUCTION

This section defines and illustrates some of the situations (or problems found in the management of concurrent transactions in DDBs. These findings resulted from maintaining the transparency to the user and making efficient and reliable the interaction with the database. The examples presented in chapter 2 are utilized to illustrate the problems found. Once these problems have been stated, the corresponding characteristics of the concurrency control mechanisms can be determined.

3.2. PROBLEMS IN THE PROCESSING OF CONCURRENT TRANSACTIONS

There are many advantages inherent in replicated databases such as improved performance, increased data accessibility, and load sharing. Studies have been undertaken to solve the problems associated with these advantages. Some of the problems are : synchronization of concurrent updates to the database, maintenance of the consistency of the database, recovery after failures , where to put the data, how

many copies to make, and where to do the query processing.. The database of a bank application is an illustrative case (situation illustrated in chapter 2).

"BANK OF THE CITY" has branches in several areas of the city and each branch keeps a file of its own accounts as well as files of the accounts of other branches. Determining the number of copies and the sites at which to place them, are not of interest now. A typical bank requires transactions, such as Fund Transfer, Balance Inquiry, Deposit, Withdrawal, etc. In order to fulfill these transactions concurrently, a control mechanism is required to avoid problems such as violation of internal consistency [KANE79,THOM78], violation of mutual consistency [KANE79,THOM79], and deadlock [DATE83,ULLM83].

A schedule S of a set of transactions T1,T2,...,Tn represents a particular order in which the actions of the transactions were performed in the system [GARC82a]. A schedule is also a sequence of actions. The simplest schedules run all actions of one transaction and then all actions of another transaction. Such one-transaction-at-a-time schedules are called serial because they have no concurrency among transactions [GRAY78]. Clearly a serial schedule does not induce inconsistency because every transaction is assumed to be individually correct; that is, each

transaction preserves the integrity of the database if executed in isolation. A schedule is "serializable" if its effect is equivalent to that of some serial schedule.

Let us see an example : The following two transactions come to the Downtown branch. Recall from figure 2.2 that the balance of the account 1-16-DT is $5000 and the balance of the account 4-16-PF is $1000.

```
        T1 : FUND_TRANSFER

T11.....READ BALANCE (ACCOUNT = 1-16-DT);
T12.....BALANCE = BALANCE - 100 ;
T13.....READ BALANCE (ACCOUNT = 4-16-PF  ;
T14.....BALANCE = BALANCE + 100 ;

        END T1


        T2 : DEPOSIT

T21.....READ BALANCE (ACCOUNT = 4-16-PF) ;
T22.....BALANCE = BALANCE + 3000 ;

        END T2
```

If we allowed any interleaved execution order for the actions of the transactions to take place, we would get different results; therefore, serious problems of database consistency could result. In the last example, if the schedule were T11,T12,T13,T21,T14,T22, then the balance of the account 4-16-PF would be $ 4000. However, if the schedule were T11,T12,T13,T21,T22,T14, then the balance of the

account 4-16-PF would be $ 1100. The true balance of the account 4-16-PF should be $ 4100. This discrepancy violates the internal consistency constraint due to the loss of an update [DATE83]. The balance of the account 1-15-DT is $ 4900, which is independently correct of the schedule executed.

It should be noted that temporary inconsistency [ESWA76] is inherent in all sequential computations, and for this reason consistency requirements cannot generally be enforced before the end of the transaction (when the transaction commits .

There may be problems in maintaining the mutual consistency [KANE79,THOM79], because when updating an account, the effect must be propagated to all the copies of the account (if a copy of the account is kept at several branches . Taking the last example, we can see that if the schedule followed by the Downtown branch were different to that used by the Pittsford branch, the final balance of the account 4-16-PF would be different at both branches. Maintenance of mutual consistency therefore, requires all sites to make the same decision for concurrently initiated conflicting updates [THOM79]. The inherent delays in communication networks make it extremely difficult for all the copies to be equal at any instant; however, they must

converge to the same state and be identical when the update process ceases.

These concurrency anomalies are very difficult to understand and guard against. Most transaction management systems therefore, hide concurrency from users by implementing techniques such as Locks, Timestamps, Circulating Permits and Tickets, Conflict Analysis, and Reservations [KOHL81]. These mechanisms solve part of the problem, but new problems arise like DEAD-LOCK, LIVE-LOCK and Overhead.

Intuitively we see that in order to preserve the consistency, some actions should wait for others to be done; thereby making sure that the transactions do not see obsolete data. This wait causes a delay and may cause a mutual block (deadlock) in the execution of transactions. To illustrate this case let us take a typical transaction. (Figure 3.1)

If the generated schedule for T1 and T2 is T11,T12,T21,T22,T13,T14,T23, T24, we can see in figure 3.2 when the deadlock occurs. The transaction T1, in order to read and update the account 1-15-DT, locks it to avoid that other transactions see it before it is updated. The transaction T2 does the same operation for the same reason T1 did.

```
        FUND_TRANSFER

        T1 : FUND_TRASNF_1

T11.....READ BALANCE (ACCOUNT = 1-15-DT) ;
T12.....BALANCE = BALANCE - 100 ;
T13.....READ BALANCE (ACCOUNT = 1-17-DT  ;
T14.....BALANCE = BALANCE + 100 ;

        END T1


        T2 : FUND_TRANSF_2

T21.....READ BALANCE (ACCOUNT = 1-17-DT) ;
T22.....BALANCE = BALANCE - 50 ;
T23.....READ BALANCE (ACCOUNT = 1-15-DT  ;
T24.....BALANCE = BALANCE + 50 ;

        END T2
```

Figure 3.1. Fund Transfers.

Now, when T1 and T2 try to execute their last actions, they find out that they can not lock the items needed. As a result, T1 is waiting for T2 to release a lock on an item it needs, and T2 is waiting for T1 to release a lock on an item it needs. If this situation is not detected, T1 and T2 may remain waiting forever.

A deadlock may occur at a local level, a branch, or at a global level, involving several branches [GRAY78]. Menasce and Muntz [MENA79] consider that there are three approaches to the treatment of deadlocks : deadlock avoidance, deadlock

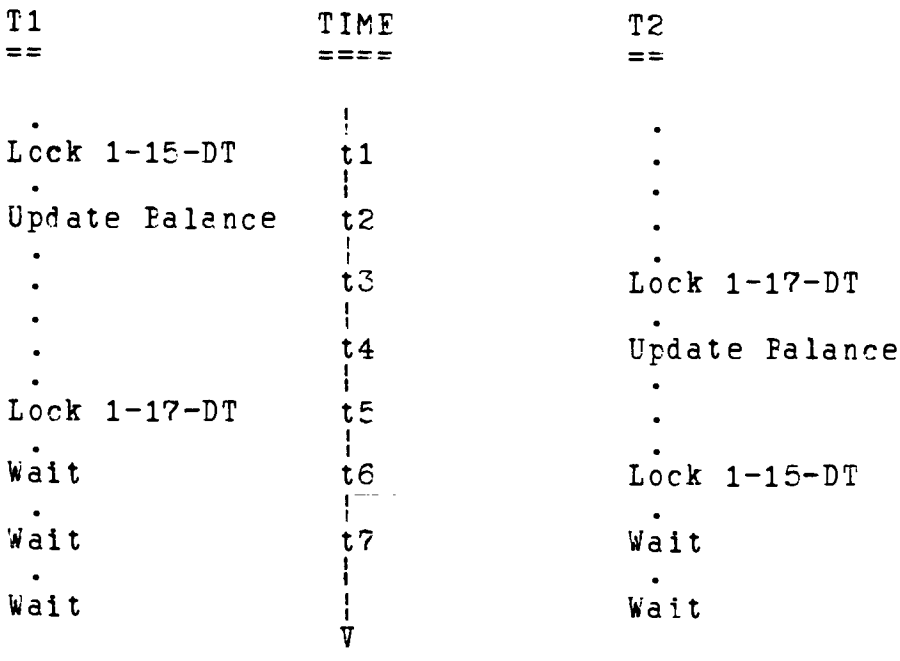


Figure 3.2 Deadlock Occurs at Time t6.

prevention and, deadlock detection and resolution; however, many authors [RYPK78,DATE83,ULLM83, BERN81] consider deadlock avoidance and deadlock prevention to be only one approach. Some of the techniques used in deadlock prevention and avoidance are: (1 to require that all the resources be acquired at once by a transaction, (2) to look ahead at what is going to happen should the lock be granted, and not honoring the lock if it is going to cause a deadlock, (3 to assign an arbitrary linear order to the resources and require all transactions to request locks in this order, 4) to use timestamps. In using timestamps no data is ever locked, thus deadlock is impossible. The second approach to

handling deadlocks is to do nothing to prevent them, and to use methods to find them after they have occurred (see [BERN81] for details about methods for deadlock detection . Breaking or resolving a deadlock consists of choosing a "victim" -that is, one of the locked transactions- and rolling it back [DATE83]. The system must guarantee that a transaction that is put on wait, will eventually come out of the wait state. When a transaction waits forever to have a lock honored, it is known as Live-Lock.

For the benefit of the users, the DBMS should provide consistent views of the data during failure (reliability [CHOU80a,MENA80a] , as well as maintain the database consistent when the failure is fixed and normal operation is resumed. For example, in the network of "BANK OF THE CITY" communications between the Pittsford branch and the rest of the branches are broken temporarily, causing the partition of the network. Should a partition happen, the system should be able to keep executing transactions, gracefully degraded though [CHOU80a,MENA80a]. This situation may lead to the following undesirable case. If the Pittsford branch were to be disconnected from the network, clients having accounts at the Pittsford branch could cheat the bank in the following way : by going to Pittsford, withdrawing all the money, then going Downtown and doing the same. As a result, the client will double his money. When communications are restored, the

system will try to propagate the updates but the client will be gone.

Network partition is one of the worst disasters that can befall a distributed data base system. Most concurrency systems depend upon prompt message passing to maintain locks or voting. Thus if a network becomes partitioned, the independent groups could continue running and build conflicting versions of the database [CHII82]. The system designer must decide how the system will operate after a partition has occurred, in order to return the database to a consistent state after the partitions merge. If necessary inconsistencies will be corrected. He or she has three basic alternatives [GARC82b]:

(1) Allow each group of nodes to process new transactions,
(2 Allow at most one group to process transactions, and
(3) Halt all transaction processing until the system is united again.

Not only should these problems be solved by the concurrency control mechanism, but also the solutions should take place without incurring, highly complex programs, excessive use of resources (overhead [BADA80]), delays and a high volume of messages (efficiency [KANE79,BERN79a, BADA80]). A good concurrency control mechanism must solve

these problems as well as others presented in more complex applications. Such a mechanism should have four important characteristics: it must be correct, efficient, reliable, and general (it is obvious that the election of a mechanism for a specific application depends upon the needs and the environment where it is going to be implemented).

An application may have its data distributed in any one of the following forms: fully replicated, partially replicated or non replicated. Non replication implies a great simplification of the concurrency problem, but the DDB loses some of its main attractions such as reliability and availability. On the other hand, full replication may be inefficient depending on the size of the database and the frequency of its use. Furthermore, redundant updating can be costly because it may potentially involve extensive intercomputer communication overhead in order to lock all copies of data being updated. Algorithms to handle specific distributions have been proposed, but they lack generality causing the reduction of their applicability. Unlike specific case algorithms, algorithms which handle partial replication are more general, but their complexity is far greater. Full and null replication can be treated as special cases of partial replication, allowing partial replication algorithms to cover all cases.

3.3. CHARACTERISTICS OF A GOOD CONCURRENCY CONTROL MECHANISM

This section defines the characteristics of a good algorithm. These characteristics are correctness, efficiency, reliability and generality.

3.3.1. CORRECTNESS

Since concurrency control is the most important part of a DDBMS, we should be sure that the algorithms developed to implement it are correct. These algorithms are usually complex, hard to understand, and difficult to prove correct (indeed, many are incorrect) [BERN81]. An algorithm works correctly as long as it meets the following requirements :

- Preserves Internal Consistency.

  The net effect of executing several transactions concurrently should be equivalent to the effect of executing them in some order, but serially [MENA79,BERN79a, ESWA76, IAMP78,BAYE80b]. Serializability is the formal criterion for correct execution of a set of concurrent transactions [DATE83]. It is based on the assumption that each user transaction will preserve database consistency if it runs atomically.

- Preserves Mutual Consistency.

All the copies of the database must converge to the same state and must be identical when the update process ceases [LAMP79,MENA80b,THOM79]. Two kinds of mutual consistency are distinguished [WILM80]: (1 strong mutual consistency: Between two updates, all copies belonging to running sites have the same version of the database, and (2 weak mutual consistency: Different sites may possess different versions at a given time.

- Deadlock Free.

The avoidance of a transaction waiting forever to be executed due to a deadlock. It should detect and solve or prevent local deadlock and global deadlock [GRAY78, MENA79,BERN79a].

- Avoids Critical Blocking.

All the transactions should be executed or rejected in a finite time [HILL77b,BERN80b].

3.3.2. EFFICIENCY

Because the concurrency control mechanism is heavily used in a distributed environment, its efficiency plays an important role in the overall performance of a DDBMS. Some of the features that increase efficiency are:

- Distinguish Type of Transaction.

Different types of transactions need different levels of synchronization. Some transactions only need local locking on a site by site basis [BERN78] and others do not require synchronization at all [BERN80b]. Having many types of algorithms avoids the overhead incurred by using complex general type algorithms. Read-only transactions could be processed with general transaction processing algorithms, but in many cases it is more efficient to process read-only transactions with special algorithms which take advantage of the knowledge that the transaction only reads [GARC82a]. When the transaction is local, the algorithm should only use the local copy [STON78,GIFF79,BERN80b].

- More Work in Heavily Loaded Sites.

  The algorithm may have a mechanism to execute more transactions issued from a heavily loaded site than from a lightly loaded site. In other words, it should avoid bottlenecks [CHOU80a,GIFF79,LE-L78].

- Good Performance [CHOU80].

  A high number of transactions executed per time unit and low delay.

- No Rejection of Transactions.

  Rejection of a transaction causes resubmission of the transaction at a later time and this increases inter-node

communication traffic and delay [CHCU80a,LE-L78].

- Low Message Traffic.

  A small number of messages should be required for synchronization and recovery.

- Incur modest computational and storage overhead in the system[KCHI81].

- Good Response Time [PADA80].

  The time required to process a transaction should be as short as possible. The mechanism should perform satisfactorily in a network environment with significant communication delay [KCHI81].

- Independence from Transmission Speed [ELLI77b].

  Speed independence allows a solution to be generally applicable to a variety of networks.

- Low Requirement of Storage for Synchronization and Recovery.

  Nodes usually keep information about updates which they have already performed, and maintain lock tables, catalogs, etc. in order to facilitate synchronization and recovery.

- Response time should be independent of the number of nodes

in the network.

- High Level of Parallelism.

The algorithm should accomplish a high level of concurrency in a node and between nodes [ELLI77b,ROTH80]. In a node, there may be intra transaction and inter transaction parallelism [WILM80]. Internal parallel processing for a given transaction requires the possibility of running simultaneously different processes dedicated to the same transaction [GRAY81].

3.3.3. RELIABILITY

Once an enterprise is committed to a computer system, it is heavily dependent on the system's ability to cope with failure. No mechanism yet developed can achieve 100 percent resiliency [BERN79a]. The degree of resiliency that can be attained depends on how much we like to pay.

- Robustness. The system should make a "best-effort" to continue service in the event that perfect service cannot be supported [AISB76a].

a) Robustness in the face of crashes of any participating site , as well as communication failures [MENA80a, CHOU80a]. The remaining live sites should remain consistent [SHAP78], and the procedures by which this is done

must not force protocols to wait for failed sites to recover before they can safely proceed [FERN80b].

b Robustness in the face of simultaneous failure of two or more participating sites [THOM79,MENA79,GIFF79, DATE83].

c) Continued local operation in the face of network partitioning [MENA80a]. Network partitions cause serious problems because it is not possible for sites to coordinate in order to ensure correct update synchronization [RAMM78]. The algorithm should operate correctly when the network is partitioned, and each partition should be able to continue with local work.

- Recovery of a Crashed Site.

The algorithm must include a recovery mechanism which must ensure the consistency of the database when a crashed site becomes operational. During the recovery process, the remaining nodes should be able to work.

- Reconnection of Partitions.

When a partition is fixed and the network reconnected, the database must return to a consistent state.

3.3.4. GENERALITY

Generality is a property which an algorithm should have because it broadens the applicability of the algorithm in different situations arising in the life of a system. A general algorithm should have as many of the following characteristics as possible:

- Acceptance of Predicate Locks.

  The locking mechanism should be able to handle the locking of a set of all entities (items) with a certain value [ESWA76,MENA80a], for example all accounts where branch = Pittsford.

- Support of temporary extra copies (replication without affecting the performance of the algorithm [GIFF79].

- Possibility of adding new nodes [BERN78].

  As the database grows in size or usage, new nodes may be added. This addition should be possible without major service disruption.

- Possibility of applying the algorithm in the case of multiple copies and partial redundancy.

- Functionality.

  Capacity of transmission of transactions and data

[ELLI77b].

- Indifference to the topology of the network [KANF79, STON79].

  The mechanism should work well for both "broadcast" networks and "point-to-point" networks.

- Few restrictions with respect to protocols of the network itself.

  It should place few constraints on the structure of the transactions [KOEL81].

- Acceptance of all types of transactions without violation of consistency.

- Homogeneity.

  The notion of homogeneity means that the database managers at all nodes perform the same control algorithms, although the nodes may have entirely different computer systems. This property lends a bit of symmetry and elegance to solutions, allows verification to proceed by viewing the control program of a single node, and leads to formal proofs of correctness of arbitrarily large networks by induction on the number of nodes in the network [ELLI77b].

Besides the characteristics aforementioned, a good algorithm should be easy to understand and easy to implement. In order to get this, it is required that the methodology used to explain the algorithm be understandable.

All the characteristics previously mentioned will form the main parameters of the analysis to be done in later chapters. It should be noted, that it is extremely difficult to find algorithms which meet all the characteristics mentioned because it is necessary to sacrifice some qualities for the sake of others. Properties of minimal message transfer, clarity of solution, elegance of solution, maximal parallelism, practicality, and generality often conflict with each other [ELLI77b]. For example, getting a good recovery mechanism may increase the storage requirements and increase the cost of communication.

CHAPTER 4

CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

4.1. INTRODUCTION

Concurrency control is the activity of coordinating concurrent accesses to data in a multiuser DBMS. Concurrency control permits users to access concurrently a database while preserving the illusion that each user is executing alone. The main difficulty in achieving this transparency is preventing work done by a user from interfering with that done by another in a way that would cause inconsistent results or an erroneous database state to occur.

The concurrency control problem is increased in a distributed DBMS (DDBMS because (1 users access data stored in different computers, and (2) a concurrency control mechanism at one computer is not instantaneously aware of the activities at the other computers.

Concurrency control has been actively investigated for several years, and the problem for centralized DBMSs is well understood. Distributed concurrency, by contrast, is in a state of turbulence. More than 20 concurrency control algorithms have been proposed for DDBMSs, during the last 6

years, and several have been, or are being, implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct. (paraphrased from [BERN81])

Different authors have proposed various classifications for these algorithms. Kohler [KOHL81] divides the algorithms into five major categories according to their main feature: locking, timestamps, circulating permit and tickets, conflict analysis, and reservations. Eventcount is another synchronization technique, which works in a similar way to that of timestamps. In [ELLI77b] eventcounts serve the purpose of timestamps. Bernstein and Goodman [BERN81] divide the synchronization techniques into the following: two-phase locking based and timestamp ordering based. They list 48 different concurrency control methods that can be constructed using the two techniques aforementioned. Another division proposed by several authors is based on the control disciplines they utilize; centralized or distributed. " One class of mechanisms involves some form of centralized control whereby all update requests pass through a single central point. At this point the requests can be validated and then distributed to the various nodes for application to the local copies. A second, fundamentally different class, embodies distributed control. The validation and application of requests is distributed among the collection of nodes in the network." (paraphrased from [THOM78])

4.2. SYNCHRONIZATION TECHNIQUES

The most commonly used synchronization techniques are: locking, timestamps, conflict analysis, circulating permit and tickets, and reservations.

LOCKING

Most solutions to the access synchronization problem are based on some explicit or implicit locking scheme [ESWA76,GRAY78]. A transaction may lock items to ensure that nobody else has access to them while in a temporary inconsistent state. A lock operation is divided into three actions: request, grant, and release. When a transaction is granted a lock, it gets exclusive access to the item until the lock is released. The two-phase locking protocol requires that a transaction first acquire all the locks and then release them. In other words, a release lock cannot precede a request lock in a transaction. The problems with locks are the possibility of deadlock and the amount of message traffic it generates. The main difference between the use of locking in operating systems concurrency control and in database concurrency control is that in database concurrency control the main concern is to avoid incorrect completions [PAPA81]. The popularity of locking is probably due to its conceptual simplicity. It has been observed however,

that more sophisticated techniques may yield a higher degree of parallelism than locking, and therefore yield better performance [PAPA81].

TIMESTAMPS

A timestamp is a unique number which is assigned to a transaction or object and is chosen from a monotonically increasing sequence. It is often a function of the time of the day [KOHI81]. Locking synchronizes the schedule of a set of transactions in such a way that it is equivalent to some serial execution of those transactions, whereas timestamping synchronizes that schedule in such a way that it is equivalent to a specific serial execution, namely, the execution defined by the chronological order of the timestamps. This is a fundamental distinction between timestamping and locking techniques in general [DATE83]. The advantage of timestamps is that no locks are set, and hence deadlock is impossible. The disadvantage is that storage costs may be increased since timestamps must be permanently stored with each data item. Conservative timestamping is a technique that does not require timestamps on data items but involves a lot of intersite communication.

CONFLICT ANALYSIS

Conflict analysis is based on an analysis of how transactions can conflict with each other and how the conflicts can be avoided. Under this technique, transactions are divided into classes, according to their read-sets and write-sets, in order determine the level of synchronization required to avoid conflict and to guarantee a serializable schedule. Whenever transactions enter the system, their class is analyzed to determine if they require synchronization or not. If they do not require synchronization, the transactions are processed without the unnecessary delay due to the synchronization. If they do require synchronization, a level of synchronization is determined avoiding the use of global locking in all cases. Bernstein et al [BERN80b] have shown that their implementation of the conflict analysis approach allows more concurrency than the classical locking protocol.

CIRCULATING PERMIT AND TICKETS

The basic scheme can be briefly described as follows: controllers (one per node are linked together to form a virtual communication ring on which a unique control token circulates. Each node is assumed to be the agent for a single user transaction. Transactions are distinguished as queries and updates. Only the controller owning the token is allowed to initiate an update transaction. When the update

is completed, the token is passed to the successor on the ring. The circulating token serializes the update requests. The drawback of this simple approach is that there is no parallelism even when transactions modify separate objects.

In order to improve parallelism the database is partitioned into r independent parts, where access to each part is monitored by a database controller. The circulating token now carries r ticket values, one for each database partition. A database controller can update its part if it owns a ticket value. (paraphrased from [KOHL81])

RESERVATIONS

" Under this technique, each active reservable database entity has a reservation list associated with it. One protocol requires a transaction to preclaim and reserve the entities it uses in exchange for the guarantee that it will not be stopped once started. The second protocol allows a transaction to dynamically request entities but does not guarantee that the transaction will be restarted in case of conflicts. These protocols use timestamps to avoid deadlocks. (paraphrased from [KOE181]

4.3. BRIEF DESCRIPTION OF SOME ALGORITHMS

In this section, we describe some of the distributed concurrency control algorithms proposed in recent years. The description is not meant to fully cover the peculiarities of each algorithm, but to show their main features. We have tried to include algorithms from all the categories into which they are classified. Five of these algorithms will be evaluated in chapter 5.

4.3.1. Thomas' Majority Consensus Algorithm [THOM79]

Thomas' algorithm assumes a fully redundant database, with every logical data item stored at every node. Each logical data item has a value and a timestamp associated with it. An item's timestamp represents the time when the item received its current value. Since the database is fully redundant, queries are performed in a straightforward way using the local copy. An update transaction proceeds according to the following steps: (1) Query Database. The transaction queries the database in order to retrieve the items required in its update computation (this set of items is called the read-set . (2 Compute Update. The transaction computes the new values of the items to be updated (this set of items is called the write-set). (3) Submit Request. The transaction submits an update request to its local DBMS.

(4 Synchronize Update. Each one of the local DBMSs votes (YES,NO,PASS, or DEFER) to decide to accept or reject the update request. (5 Apply Update. If the request is accepted, each local DBMS applies the update to its local copy. (6 Notify. A DBMS notifies the node which generated the transaction of how the request was solved.

It is only required that a majority of nodes OKAY the request in order to approve it; however, a NO vote is enough to reject the request.

4.3.2. Menasce, Popek and Muntz [MENA80a]

The algorithm proposed by Menasce et al uses locking in order to synchronize concurrency. The algorithm has as its core a centralized locking protocol with distributed recovery procedures. A particular node is chosen to have the centralized lock controller (LC). All lock and lock release requests are routed to the LC. The LC is responsible among other things for examining lock and lock release requests coming from transactions, and deciding whether or not they should be granted. It is also encharged with detecting and resolving deadlocks. The LC maintains a table called the LOCK table, which is a set of all the active locks in the network. At the remaining nodes there is a local lock controller (LLC), which is responsible for maintaining a local

copy of the relevant portion of the LOCK table. In other words, the LLC maintains a lock table of the locks in the node it controls.

The operation of the locking protocol under no crash conditions can be intuitively explained as follows. The LC receives lock and lock release requests and decides whether or not the lock can be granted. If there are no conflicts, the request is then sent to all relevant LLCs. An LLC is relevant if its node stores data needed by the request in question. The LLC stores the request in its pending list and sends back an acknowledgement to the LC. After the LC has received the acknowledgement from all the relevant nodes, it sends a confirmation of the request to all the relevant LLCs causing the request to be transferred from the pending list to the lock table.

The basic structure of the lock and release granting algorithms is the SafeTalk protocol, which is an optimized variant of the two-phase commit protocol described in [GRAY78].

4.3.3. Bernstein, Shipman and Rothnie's SDD-1 [BERN80b, ROTH80]

SDD-1 is the first general-purpose DDBMS ever developed. SDD-1 is implemented for DEC-10 and DEC-20 computers running the TENEX and TOPS-20 operating systems; its communication medium is the ARPA network. SDD-1 is built on top of existing software to the extent possible: most notably it employs an existing DBMS, called Data Computer, to handle all database management issues. The synchronization techniques used by SDD-1 are quite different from locking. The first mechanism, called Conflict Graph Analysis, is a technique for analyzing "classes" of transactions to detect those transactions that require little or no synchronization at all. The second mechanism consists of a set of synchronization protocols based on timestamps, which synchronize those transactions that need it. The transaction classes are defined by the database administrator when the database is designed. Every physical data item in the database is timestamped with the time of the most recent transaction that updated it. Transactions also carry a timestamp which indicates the time when they originated.

The read-set of a transaction is the portion of the database it reads, and its write-set is the portion of the database it updates. Two transactions conflict if their read-sets or write-sets intersect each other. The conflict analysis is done when the database is designed and transaction classes are used instead of transactions.

In SDD-1 the execution of a transaction is supervised by a transaction manager (TM and proceeds in three phases called read, execute, and write phase.

In the read phase, the transaction is analyzed and its read-set determined. Then, the TM selects a class in which the transaction fits, and by examining class conflicts it can determine the amount of synchronization required by each transaction. Since the database is generally partly redundant, the TM chooses which copies of the read-set to read. The reading of the read-set is done by sending READ requests to those nodes at which the selected copies are stored.

During the execute phase the TM supervises the execution of the transaction. The output of this phase is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of queries). This output list is produced in a workspace, not in the permanent database.

In the write phase, the output of the second phase is broadcast to all relevant nodes as WRITE requests. If the transaction was a query, the retrieved data is displayed to the user; otherwise, each relevant node updates its portion of the write-set.

4.3.4. Kaneko's Logical Clock Method [KANE78]

Kaneko et al. present an update synchronization method which introduces logical clocks to provide database management systems with timing for updating duplicated databases. Each host has a logical clock which runs synchronously with clocks residing in other hosts. The logic clocks run in discrete 'ticks' rather than continuously. Two methods are presented. The first one requires two "ticks" of the clock to process an update command. This solution has sufficient flexibility to be modified according to requirements, and it is capable of adopting a lock and unlock scheme, or a timestamp scheme, for preserving internal consistency.

The second scheme requires two sets of messages: the first one to verify conflicts and the second one to do the update. Even though there are more messages, the exchanged message volume is reduced. It requires four 'ticks' of the clock to process an update command. Failures are easily detected due to the rules established for synchronization of the clocks.

4.3.5. Badal and Popek [BADA78]

Badal and Popek present two distributed concurrency control methods for partially redundant distributed database systems. The response time is independent of the number of sites (nodes) in the network. The synchronization protocols

are based on associating timestamps with transactions rather than with data items and assuming that all interfering actions of transactions at any given network site are executed in order of their timestamps.

The first protocol can be briefly outlined as follows. When a transaction arrives at a particular site, its read-set and write-set are determined. The initiating site determines the nodes relevant to the transaction and sends a setup message to each relevant node. The setup message contains the definition of the transaction, the list of sites involved, objects to be accessed, and a timestamp. One of the sites where a read will be done is chosen to coordinate the execution of the read and write actions. This site communicates with all other sites in the network to avoid conflicts among concurrent transactions.

The second protocol is an optimization of the first protocol. It minimizes the number of messages required by combining together two of the messages used on the first protocol. The performance of these protocols under abnormal conditions is very similar, but under normal conditions, the second protocol performs better than the first one.

4.3.6. Stonebraker's Distributed INGRES [STON79]

Stonebraker proposes an algorithm based on what is called the 'primary site' model for keeping copies. Each item possesses a known "primary" node to which all updates in the network for that item are first directed. Different objects may have different primary nodes. Locking is handled in a distributed fashion; however, deadlock detection and resolution is handled in a centralized fashion.

When a transaction originates from a user process at some node in the network, a 'master' or coordinating collection of INGRES processes is invoked at that node. This "master' creates 'slave' INGRES processes at the relevant nodes (nodes where processing will take place . The master can send two commands to slaves: (1) run a local interaction at a subset of the nodes, and (2) send a copy of the data to a subset of the nodes in the network. The slaves eventually return a 'done' to the orders of the master.

A local concurrency controller (LCC runs at each node and is encharged with handling locks at local level. If a global deadlock is detected, the node which detected it sends a message to a special node called the "SNOCP". The SNOCP then detects the deadlock by ordinary analysis of the global WAIT-FOR graph.

This concurrency control mechanism is used in the distributed database version of INGRES. INGRES is a relational DBMS which operates as a collection of user processes on top of the UNIX operating system [STON77].

4.3.7. Gifford's Weighted Voting Algorithm [GIFF79]

Gifford proposes an algorithm for maintaining replicated data, where every copy of a replicated file (files are arrays of bytes addressed by read and write operations) is assigned some number of votes. Each transaction collects a read quorum of "r" votes to read a file, and a write quorum of "w" votes to write a file, such that "r+w" is greater than the total number of votes associated with the file. This ensures that there is at least one copy which votes for read and write. Each copy has a version number which is used to determine the currency of the copies. For example: the network is formed by three nodes, each one keeping a copy of file 'A'. The file 'A' has been assigned the votes '2,1,1' in each node respectively, read votes "r" = 2, and write votes "w" = 3 (2 + 1 + 1 > 2 + 1 + 1). If a read request comes from node 1 (votes = 2), it can be satisfied locally; however, if a write request comes from any node, it must access node 1 (votes = 2) and one of the remaining nodes (votes = 1) to collect the three votes required for a write.

To read, a read quorum "r" must be gathered to ensure that a current representative of the item being read is included. From the quorum, any current representative can be actually read. To write, a write quorum "w" must be gathered: all of the representatives in the quorum must be current in order to avoid updating obsolete copies. All of the writes to the quorum are done in parallel.

The performance and reliability of the system can be tuned by manipulating the voting configuration of the files.

4.3.8. Ellis' Algorithm [ELLI77a,ELLI77b]

C.A. Ellis proposes an algorithm which, in his words, has all the desirable properties of an algorithm; such as speed independence, deadlock freedom, high parallelism, functionality, robustness, etc. Five solutions are presented: a centralized one, a sequential one, a parallel decentralized one, a queue-oriented one, and a ring structured one, but they do not comply with all the desirable properties. The database is assumed to be fully replicated.

In the centralized solution [ELLI77a], a user wanting to update the database contacts his local controller, which in turn communicates with the central controller. If no other node is updating the database, then the central con-

troller grants permission and the update is broadcast to all nodes. When the update is done at the originating node, the other nodes are allowed to update. Under this mechanism there is no concurrency of transactions.

In the decentralized solution [ELLI77a], a user wanting to update the database, contacts its local controller, which in turn broadcasts the request directly to all other database nodes. It must then wait until it receives positive acknowledgement from all other nodes. If any node responds with a negative acknowledgement, then the request is aborted. If all acknowledgements are positive, then the update is done locally and the updated data is broadcast to all other database nodes. Finally, the nodes do the local updating and acknowledge it.

The queue-oriented solution [ELLI77b] is presented as a solution to some of the problems of the previous alternatives. It tries to avoid critical blocking by forming a queue of all requests which occur near the same time point. This method offers a decrease in the amount of messages transmitted when the network is heavily loaded with updates because one locking allows a whole queue of updates to take place.

The ring structured solution [ELLI77b] requires that the communication medium be a ring, so each node can only communicate with its successor on the ring. To update any copy of the database, a transaction must first lock the entire database at all nodes. A request to update propagates around the ring structure and is accepted when it returns to its sender. The update is done in a similar manner. This solution forces all transactions to execute serially. It is deadlock free and does not reject requests.

4.3.9. J.C. Seguin et al [SEGU79]

Seguin et al present a majority consensus algorithm for fully duplicated databases. A set of duplicated copies are distributed among several hosts on a network. A monitor is attached to each host owning a copy. The information is global to all the monitors and cannot be partitioned, so independent subnetworks cannot operate simultaneously; therefore, a majority consensus of monitors is searched before each update. This synchronization step among the monitors prohibits simultaneous handling of several requests. Each monitor possesses an actuality degree which is a timestamp associated to the copy and characterizing the currency level of this copy. This timestamp is used to synchronize updates.

Each monitor is characterized by one of the following states. ADMINISTRATOR: privileged monitor owning the original copy. FELLOW: monitor connected to the administrator, and receiving the updates sent by the administrator. POSTULANT: monitor coming from the state of fellow, and wanting to initiate an update.

Every copy associated with a monitor possesses one of the following states. STABLE: there is no modification being processed on the copy. UNSTABLE: an update is being processed or the copy. A new update request cannot be initiated.

When some monitor (fellow receives a local request, it asks the administrator the authorization for initiating an update. After getting the authorization, it becomes postulant. The administrator accepts the demand and takes the fellow state. When all the monitors are informed that the postulant would become administrator, and after they have sent back an acknowledgement to the postulant, this one becomes administrator.

The new administrator sends all the fellow monitors the new version of the copy and the new actuality degree. Its copy becomes unstable because a modification is being performed. When a fellow monitor receives the new version, it sets its copy to unstable and adjusts its actuality degree.

Then it acknowledges the receiving of the new copy.

When the administrator has received a quorum of acknowledgements, it sends all the monitors the order to process the update, it locally executes the update, and its copy becomes stable again. Each fellow monitor receiving the order to update its copy processes the same operation on its copy.

4.3.10. G. Le Lann's Tickets Algorithm [LE-L78]

Le Lann proposes a distributed approach based on a circulating permit or control token on a virtual ring. Each node of the network is assigned a controller. Controllers are linked together on a virtual ring and a unique specific message, the control token, circulates on the ring. Only the controller owning this token is allowed to start a specific activity. Upon completion, the token is given to the successor on the ring. The drawback of this approach is the complete serialization of transactions which allows no parallelism.

Le Lann proposes some extensions of this simple approach. The extensions improve the parallelism allowed. In this case the database is assumed to be partitioned into r independent parts. One or several token may circulate on the ring, and each one carries r ticket values, one per partition.

An update transaction must request and obtain the next consecutive ticket for each partition it will access before proceeding. Tickets can only be obtained from a control token when it visits the node on which the transaction is running. The circulating token no longer needs to wait for the transaction to complete before being passed to the successor on the ring, since access is serialized by the consecutive tickets. Every controller is assumed to take care of deadlocks possibly existing between its local processes.

4.3.11. Chuen-Pu Chou and Ming T. Liu [CHOU80a]

Chou and Liu propose an algorithm which uses distributed control, does not reject transactions, does not use global locking, prevents deadlocks, uses timestamps (not to label database data), and the database is assumed to be fully redundant. A consideration of this algorithm for the partially replicated database case is described in [CHOU80b].

A query is processed in a straightforward way. The progress of an update transaction is divided into three phases: transmission, selection, and process.

Transmitting phase : After the consistency enforcer receives a transaction, a priority is assigned to the tran-

saction. Then the transaction is broadcast to all the nodes in the network. When a node receives the transaction, it is put in the Execution Waiting Queue (EWQ).

Selecting phase : Transactions waiting in the EWQ are selected by the consistency enforcer and then dispatched to the local DBMSs for processing. Priorities and dummy transactions are used to ensure mutual consistency and to avoid unnecessary waits.

Processing phase : After a transaction is sent to a local DBMS for processing, the concurrency control problem is changed from a distributed environment to a centralized environment. It is in this phase when transactions are processed concurrently at the local level.

4.3.12. Wing K. Cheng and Geneva G. Belford [CHEN80]

Cheng and Belford present a distributed algorithm for update control whereby the update initiating node acts as a semi-centralized manager for that update. Timestamps are used to mark the transactions and are not stored with each of the data items of the database. The database is assumed to be fully replicated. Comments on how the algorithm can be modified to handle partially replicated databases are presented.

Each site of the network maintains two types of queues to hold uncommitted requests: the queue QLOCK for local requests and a queue QFOR for requests originating at every foreign site. Messages in each queue are ordered according to their timestamps. The algorithm essentially consists of three phases.

During the first phase, the local lock tables are examined to determine if the request conflicts with the requests already in the system. If the request conflicts, it is rejected. Otherwise, the read-set is read and the new values computed.

Second phase. The site originating the request sends a message containing the write-set (including the new values) to every site in the network. Then it waits for a message from each site to determine whether the update should be committed or aborted. A COMMIT or ABORT message is then sent to every site.

In the third phase, every site receives either a COMMIT or ABORT message from the originating node. If the message is COMMIT, the update request is committed by the local DBMS, and an acknowledgement is sent to the the originating site. On the other hand, if the message is ABORT, the update is aborted.

4.4. CONCLUSIONS

It should be noted that the authors of the described algorithms have utilized different methodologies to design and explain their work. Methods such as evaluation nets (they are a modified form of Petri nets [ELLI77b], extensions of concurrent languages (MESA is a language developed at the Xerox Palo Alto Research Center) [GIFF79], and completely narrative descriptions are some of the methods used for explaining the algorithms, and as such, are far from being adequate tools. This has made it difficult to thoroughly understand the algorithms and implement them.

Analyzing the different features of the algorithms, their clarity, the amount of information available, their speciality in concurrency control, and the different techniques used, we have chosen the following algorithms for the evaluation :

R. Thomas [THOM79]

Chuen-Pe Chou and Ming T. Liu [CHOU80a]

D.A. Menasce, R. Muntz, and G. Popek [MENA80a]

C. Ellis [ELLI79b]

M. Stonebraker [STON79]

CHAPTER 5

QUALITATIVE AND QUANTITATIVE EVALUATION OF THE ALGORITHMS

5.1. INTRODUCTION

The algorithms previously described utilize various synchronization techniques in order to maintain the consistency of the database. Moreover, every algorithm is built upon different assumptions about the underlying DDBMS, such as network topology, frequency of every type of transaction, data replication, reliability of computer hardware, etc.

The use of different techniques and the different assumptions made, make it difficult to compare and analyze all the algorithms on the same base.

Currently, the power and goodness of new mechanisms are demonstrated by showing solutions to a number of standard synchronization problems. Therefore, it is clear that some well-defined methodology for evaluating synchronization mechanisms is needed.

This chapter includes a summary of some previous work related with synchronization mechanisms evaluation, and sets forth some criteria to be used to compare these mechanisms.

The qualitative and quantitative criteria are the base for the elaboration of a comparison and classification scheme.

5.2. SUMMARY OF PREVIOUS WORK

P. Wilms [WIIM80] has established some criteria which should be fulfilled by concurrency control algorithms depending on the kind of application under consideration. These criteria are commented and the principal characteristics of some algorithms are summarized. Wilms' study also includes some parameters and distribution laws which could be taken into account for the quantitative comparison of these algorithms.

W.K. Lin [LIN-81] evaluated two concurrency control mechanisms in terms of protocol synchronization delays and average transaction response time by using simulation. The mechanisms evaluated are SDD-1 [BERN80b] and Dynamic Timestamp Method (Lin's own mechanism).

Garcia-Molina [GARC78] analyses and compares the performance of two algorithms in the case of completely duplicated databases in a no failure, update only environment. One of the algorithms is a centralized locking algorithm while the other is distributed voting algorithm. The algorithms are studied through detailed simulations. The results obtained were also approximated by an analytic technique

based on a queueing model. The parameters studied in these simulations include mean interarrival time of updates, number of items in the database, number of nodes in the network, network transmission time, CPU time and IO time. The results of these simulations show that the centralized locking algorithm performs better than the distributed voting algorithm except in cases of extreme IO utilization.

Kaneko et al. [KANE79] use an event driven simulator called DDACS which was developed for evaluating distributed database control methods. A performance comparison is made between a logic clock synchronization method (one proposed by Kaneko et al. and other methods. A centralized control method and a majority consensus method are compared to the logical clock method. The results of this simulation are similar to those of Garcia-Molina's even though kaneko et al. concluded that their method performs better than the centralized method. An important observation presented in Kaneko et al. is the influence that the network topology has on the performance of the algorithms.

The use of simulation models presents some problems. Since all the characteristics of the algorithms are not included in the analytic models used for the simulation, the results cannot be safely used to determine which mechanism is better. However, they produce insights useful to the designer of distributed databases. It should be noted that

these simulation studies are specific and do not cutline a methodology for evaluation.

Based on the previous work, this chapter uses some evaluation methods similar to those utilized in [WILM80].

In this chapter, we will try to evaluate some parameters which do not depend on a specific type of application, but rather depend on the algorithm itself. These parameters include the number of messages required for an update and the number of messages required to recover a crashed node or join a network partition.

This chapter also includes an evaluation of those parameters which are dependent on a specific application. This evaluation is done in a standard way for all the algorithms and includes the most significant parameters.

5.3. QUALITATIVE EVALUATION

With the definitions and characteristics proposed in chapter 4 used as headings, we have made a comparative chart of the algorithms chosen in chapter 4. The information contained in the chart is either taken or inferred from the following references: [PERN80b], [CHOU80a], [ELLI79b], [MENA80a], [ROTE80], [STON79], [THOM79], [WILM82].

Figure 5.1.

CHARACTERISTIC \ ALGORITHM	THOM	MFNA	ELLI	CHOU	STON	BERN
MUTUAL CONSISTENCY						
- Strong		y	y	y	y	
- Weak	y				y	y
MUTUAL CONSISTENCY						
- Centralized obtainment		y			y	
- Distributed obtainment	y		y	y		y
DEADLOCK HANDLING						
- Avoidance & Prevention	y	y	y	y		y
- Detection & Resolution					y	
TRANSACTION TREATMENT						
- Treat Read-only specially		y			y	y
- Treat all the same	y		y	y		
POSSIBILITY OF BOTTLENECKS	y	y	y	n	n	y
RELIABILITY						
- Failure of one node	y	y	y	y	y	y
- Failure of several nodes	y	y	y	y	y	y
- Network partition	-	y	n	y	y	y
NETWORK PARTITION						
- Subnets work independently	-	y	-	y	y	-
PREDICATE LOCKS ACCEPTANCE	n	y	-	y	y	y
DATABASE						
- Full replication	y	y	y	y	y	y
- Partial replication	n	y	n	ext	y	y
- No replication	n	y	n	ext	y	y
PARALLELISM						
- Intra transactions	n	-	n	-	-	y
- Inter transactions	y	y	n	y	y	y
LOGIC COMMUNICATION						
- Broadcast	y	y		y	y	y

- Virtual ring	y		y			
PHYSICAL COMMUNICATION						
- Reliable broadcast				y	y	y
- Non reliable broadcast	y	y		y	y	
- Point-to-point			y	y	y	y
COMPLEXITY	h	m	l	l	l	h
HOMOGENEITY	y	n	y	y	n	y
SYNCHRONIZATION TECHNIQUE						
- Timestamp	y			y		y
- Eventcount			y			
- Locking		y			y	
- Conflict Analysis						y
METHOD OF CONTROL						
- Centralized		y			y	
- Distributed	y		y	y	y	y
BEFORE CONFLICTS						
- Reject	y	y				
- Wait	y		y		y	
- Serialize				y		y
AUTOMATIC RECOVERY	y	y	y	y	y	y
PROCESSING INTERRUPT DURING RECOVERY	y	y	n	n	y	n

Figure 5.1 Qualitative Evaluation

Note : - Ellis' ring solution is the one being evaluated.

- "y" means that the algorithm has the property or characteristic.
- "n" means that the algorithm does not have the property or characteristic.
- "-" means that there is not enough information to determine to determine the existence of properties

or characteristics.

- "ext" means that the author has proposed an extension which includes this characteristic or property.
- "h" means high.
- "m" means medium.
- "l" means low.

A reliable broadcast ensures that the totality or none of the connected sites receive the message. In other algorithms, the ackrowledgement of the totality of sites is necessary in order to be sure that the broadcast has been correctly performed.

While Menasce's algorithm allows all the network partitions to keep working, Chou's and Stonebraker's algorithms only allow only one partition to keep working. The implementor is responsible for selecting the partition which will be allowed to remain working.

It is difficult, and perhaps a waste of time, to give an accurate evaluation of these algorithms without taking into account the precise environment where the algorithm will work. Indeed, the interest of an algorithm depends on

[WILM82]: (1) the kind of transactions to be processed, and their distribution, (2) the arrival rate of the transactions, which influences the frequency of conflicts, (3) response time required, (4 replication of the data in the database, and (5) reliability of the communication subnetwork.

5.4. QUANTITATIVE EVALUATION

This evaluation only considers the behavior of the algorithms in the case of fully replicated databases.

5.4.1. PARAMETERS

In order to evaluate each algorithm in the quantitative aspect, it is necessary to define and list some typical parameters which influence the performance of the algorithms.

a) UPDATE TRANSACTIONS.

The mean interarrival time of update transactions at each node is ARU. It is assumed that the arrivals of update transactions follow a Poisson distribution. Therefore, the average update transactions arrival rate is LU = 1 / ARU : it represents the number of update transactions that arrive per second. The average update transactions

arrival rate includes local transactions and those coming from other nodes.

This arrival rate is :

$$LU = \sum_{J=1}^{N} LU_J$$

$$ARU = 1/LU$$

b) READ-ONLY TRANSACTIONS.

It is assumed that read-only transactions only affect the site where they are initiated. The mean interarrival time at each node is ARR. This distribution is also assumed to be Poisson with a mean value ARR.

The arrival rate is:

$$LR = LR_{LOC}$$

$$ARR = ARR_{LOC} = 1 / LR_{LOC}$$

where LOC is the node where the transaction is initiated.

c) DATABASE.

DA : Distribution of the accesses to the database items. This access distribution is highly correlated with the application. It refers to the mean number of items referenced by a transaction.

M : Total number of items contained in the database.

Pc : Probability of conflict between two concurrent transactions. If we assume that every item of the database is lockable, and DA1 and DA2 are the number of items referenced by two transactions respectively, then [WILM80]

$$Pc = 1 \quad \text{if} \quad DA1 + DA2 > M$$

$$Pc = 1 - \text{Probability-no-conflict}$$

$$\text{Prob-no-conflict} = \frac{\binom{M - DA1}{DA2}}{\binom{M}{DA2}} \quad \text{(see MEYE65}$$

$$Pc = 1 - \frac{(M-DA1)! * (M-DA2)!}{M! * (M-DA1-DA2)!}$$

$$\text{if } DA1 + DA2 <= M$$

d) NETWORK.

N : This number represents the number of nodes in the network. These nodes are divided into two categories: the nodes which are active (NA , and the nodes which are down (ND).

$$N = NA + ND$$

TA : Average time that a node is active (up).

TC : Average time that a node is crashed (down .

Pf : Probability of a node being crashed.

$$Pf = TC / (TA + TC$$

Pa : Probability of a node being active.

$$Pa = 1 - Pf$$

P(NA) : Assuming that node failures are independent of each other, the probability of having NA nodes active is the probability that NA events happen, given that there are N repetitions (Binomial distribution [DRAK67, MEYE65] .

$$P(NA) = \frac{N!}{NA!(N - NA)!} (1 - Pf)^{NA} * Pf^{(N - NA)}$$

Failures in the network links should be included, but it is difficult to calculate how many nodes become isolated because this depends on the network topology.

TT : Average network transmission time. We could assume that the time it takes for any message to go from any node to any other node is constant, but it is only acceptable if the load is light or uniformly distributed. We can calculate this

value as the average transmission time between any pair of nodes in the network.

$$TT = \frac{\sum_{i=1}^{N} \sum_{J=1}^{N} TT_{i,J}}{N(N-1)} \qquad i \neq J$$

where $TT_{i,J}$ = average transmission time from node i to node j.

e) PROCESSING and IO TIME.

CIT : Time it takes to compute a new value for a given item. If x is the number of items referenced by a transaction, the total computation time is x * CIT.

TS : Time it takes to do a small computation such as comparing two values, set or release a lock, etc.

IOT : Time it takes to read or write a value from or to an IO device.

5.4.2. MEASUREMENTS

The measurements that we try to find in this section depend on the parameters previously listed, and are aimed at evaluating the efficiency of each algorithm during normal

operation and when recovering a crashed node.

1. NUMBER OF MESSAGES.

This section refers to the number of messages required for an update to take place when no conflict occurs (NMSU), the number of extra-messages resulting from a conflict (NMC), and the number of messages generated during a recovery process (NMR).

The general equation of the average number of messages required by an update transaction is [WIIM80]:

NMU = NMSU + Pcc * NMC * NR

where Pcc is the probability of conflict :

Pcc = Pc * Ps

Pc = probability of conflict in case of concurrent transactions

Ps = probability of concurrency

NR = number of repeated rejections on the same transaction

2. LOAD OF A NODE.

The load of a node depends on the distribution of the transactions among the different nodes and the technique used for concurrency control. If the technique used is a centralized one, the load of the central node is higher

than the load of the other nodes; therefore, a bottleneck can result at the central node.

3. PROPORTION OF UPDATE TRANSACTIONS (PU).

It is the ratio between the average of update transactions and the average of transactions (update + read-only .

PU = LU / (LU + LR)

4. UPDATE RESPONSE TIME (UTresp).

The resporse time of a transaction is the difference between the finishing time and the time when the transaction arrived at the initiating node. A transaction is finished when the user is notified that its transaction was processed.

The general formula for the update response time is [SEGU79, WILM80]:

$$UTresp = IT + TRc * Pcc + z * TT + y * (CIT + ICT) + x * (TS + ICT)$$

where : IT is the initialization time,

TRc is the average time for conflict resolution,

y is the number of items implicated in the updat transaction.

z is the number of non simultaneous messages

required for the update transaction, and

x is the number of elementary operations.

The general formula for read-only response time is :

$$RTresp = UTresp - z * TT$$
$$= IT + TPc * Pcc + y * (CIT + IOT)$$
$$+ x * (ST + ICT)$$

Recall that we are considering the fully replicated case, therefore no messages are required.

5. MEAN THROUGHPUT AT A NODE (MT).

It is the average of transactions performed per time unit at a node.

$$MT = \frac{1}{PU * UTresp + (1 - PU) * RTresp}$$

This formula is applicable if read-only transactions are serialized during the processing of update transactions.

6. FAILURES.

If the criteria for performance are the number of messages and the update response time, some of the described algorithms do not have a high performance in a completely

reliable environment. This is often due to the fact that, for robustness, these algorithms require extra-messages and additional storage even in the absence of failure. The impact of failure on the performance of the algorithms can be evaluated in the following terms.

a Number of messages required by a recovery procedure (NMR).

b Size of extra-information required for increasing robustness, i.e. management of journals, pending updates tables, up-nodes tables, etc.

c) Recovery time delay. It is the time lapse between the node being repaired and the point when it is fully integrated into the system.

5.4.3. EVALUATION

5.4.3.1. Number of Messages

Number of messages = NMSU + Pcc * NMC * NR

NR depends on the probability of conflict and on the rule of priority assignation for rejected transactions.

The measurements of NMSU and NMC of the studied algorithms are:

a) ELLIS.

NMSU = NA (the request propagates around the ring)
+ NA (the update propagates around the ring)
= 2 * NA

NMC = 0 (when conflict occurs the transaction waits .

b) THOMAS (Communication discipline is Daisy Chaining).

NMSU = [N / 2] (to achieve a consensus , best case
+ (N - 1) (nodes acknowledge the acceptance)
+ 1 (to notify the user
= [N / 2] + N

NMSU = N (to achieve a consensus , worst case)
+ (N - 1) (nodes acknowledge the acceptance)
+ 1 (to notify the user)
= 2 * N

In this case [x] means the smallest integer >= x.

NMC = 1 (the first node rejects , best case
+ 1 (to notify the user)
= 2

NMC = NA (the last node rejects , worst case
+ (NA - 1) (to acknowledge the rejection)
+ 1 (to notify the user)
= 2 * N

These calculation are done assuming that all nodes are active. In case some nodes are down, N is NA (active nodes .

c) MENASCE, POPEK AND MONTZ.

```
NMSU =   1          ( send lock request to the central
                      lock controller )
       + NR         ( phase 1 of lock request
       + NR         ( phase 2 of lock request )
       + NR         ( phase 1 of lock release )
       + NR         ( phase 2 of lock release )
       + 1          ( to notify the user )
     = 2 + 4 * NR
```

NR is the number of nodes relevant to the update transaction. In the case we are considering, the fully redundant database, NR = NA - 1. All nodes are relevant.

```
NMC =  1            ( the central lock controller
                      rejects and notifies the
                      initiating node )
```

d STONEBRAKER.

```
NMSU =   NR         ( master sends update request to
                      all relevant nodes
       + NR         ( relevant nodes answer "I'm ready"
```

```
        + NR            ( master sends 'commit' to all
                          relevant nodes
        + NP            ( relevant nodes answer "done" )
        + 1             ( send "done" to the user
        + 1             ( send "done" to the SNOOP )
      = 2 + 4 * NP
```

In the case of fully redundant databases, NR = NA - 1.

```
NMC =   1               ( the slave sends a 'reset' to the
                          master or to the SNOOP
```

e) CHOU AND LIU.

NMSU depends on the types of protocols used :

(i) Multi-destination protocol.

```
NSMU =   1              ( to broadcast the request to
                          other sites
       + ( NA - 1 )     ( every site sends an ACK )
       + 1              ( to notify the user )
     = NA + 1
```

(ii) Point-to-point protocol.

```
NMSU =   NA - 1         ( to send the request message
                          to other sites. ACK to the sender
                          can be piggybacked on the next
                          request message
       - 1              ( to notify the user )
     = NA

NMMC = 0
```

5.4.3.2. Load of a Node

Knowing the number of messages received and handled at each node (generated by an update transaction , we can calculate the ratio of load distribution between the node that handles the highest number of messages (Hm , and the node that handles the lowest number of messages (Lm). This ratio (CL = Lm / Hm) indicates the degree of control distribution with respect to messages of an algorithm. The more distributed the load (CL near 1), the lower the risk of bottleneck (CL << 1 at a given node.

a) ELLIS.

Lm , Hm = 1 (the node receives the request message from its predecessor and forwards it to its successor)

+ 1 (the node receives the update message from its predecessor and forwards it to its successor)

= 2

CL = 2 / 2 = 1

b) THOMAS (Communication discipline is Daisy Chaining).

Hm = 1 (the node receives the update request and forwards the request and the votes along to another node

+ 1 (the node is notified how the request was solved and forwards the notification to another node)

= 2

Lm = 1 (the node is notified how the request was solved and forwards the notification to another node. This node did not vote)

CL = 1 / 2

c) MENASCE, POPEK AND MUNTZ.

Hm = 2 + 4 * (NA - 1) (this node is the central lock controller

Lm = 4 (four messages generated by the two-phase lock protocol

$$CL = \frac{1}{2 + 4*(NA - 1)} \cong \frac{1}{(NA - 1)}$$

d STONEBRAKER.

Hm = 2 + 4 * (NA - 1) (this is the initiating node. The master is invoked at this node

Lm = 4 (four messages generated by the two-phase lock protocol

$$CL = \frac{4}{2 + 4*(NA - 1)} = \frac{1}{(NA - 1)}$$

e CHOU AND LIU.

(i) Multi-destination protocol.

Hm = 1 (to broadcast the request message to other sites)

Lm = 1 (to send an ACK)

CL = 1 / 1 = 1

(ii) Point-to-point protocol.

Hm = NA - 1 (to send the request message to other sites

Lm = 1 (to send an ACK)

CL = 1 / (NA - 1)

According to these values, Ellis' and Chou and Liu's

algorithms are the most distributed with respect to the handling of messages. It is also possible to calculate an approximation of the number of messages handled by each node, if we use the average update transaction arrival rate (LU).

Messages = LU * Em (worst case)

Messages = LU * Im (best case)

5.4.3.3. Update Response Time

Of all the values which comprise the response time (TRc, Pcc,x,z,ICT,y,CIT,TT , the total transmission time (z * TT) is the most dependent on the algorithm.

a ELLIS = NA * TT

b) THOMAS = (NA / 2) * TT (best case)

= (NA + 1 * TT (worst case

c) MENASCE et al = 3 * TT

d) STONEBRAKER = 3 * TT

e CHOU AND LIU = TT * &

Note. & : In Pou and Liu's algorithm it is difficult to calculate the exact time because it also depends on other nodes and dummy transactions.

5.4.3.4. Recovery from Failures

In this section we show some aspects of the recovery process assuming the following:

1) crash of a single node.

2 no other node crashes while one is recovering.

3) loss of memory.

a) ELLIS (Parallel Decentralized Solution).

- NMR = 1 (the node sends a "HISTORY REC" message to a host)
  + 1 (the host sends the history array)
  + 1 (the node indicates that the update is done)
  + 1 (the host communicates that the node is up again)
  + 1 (the host sends recent updates and "done" to the recovering node)
  = 5

- If another node is detected crashed, no extra-messages are generated.

- Interrupt for recovery.

  After a node is repaired, it contacts any operative node to obtain the history of updates which were missed. The contacted node which is called the host node, may continue performing updates while the other one is recovering.

Nodes blocked = 0

Interrupt time = 0

- To detect a crash.

When a crash is detected, the detecting node updates its up-nodes list and communicates the failure to the other nodes.

Nodes concerned = NA

Interrupt time = TS

b) THOMAS.

- NMR = (NA - 1) (the node sends a "I'm recovering" message to all nodes)
  + (NA - 1) (every node sends an ACK)
  + (NA - 1 (the node asks for the updates it has missed)
  + (NA - 1) (every node sends updates)
  = 4 * (NA - 1

- Interrupt for recovery.

When a node is notified that another is trying to recover, it must acknowledge this, and in addition, temporarily stop forwarding unresolved request on which the recovering node has voted, to other nodes.

Nodes concerned = NA

Interrupt time = 2 * (TT + TS)

- To detect a crash.

The algorithm does not require the database system to detect component malfunctions or outages.

c MENASCE, POPEK AND MUNTZ.

There are two cases : (i) if the Lock Controller (LC) crashes or becomes unavailable, a recovery mechanism takes place to choose a new Lock Controller, and (ii) a single node recovers from a crash.

(i) Single node.

NMP = 1 (the node sends a "Who is the LC message
+ 1 (some node answers)
+ 1 (the recovering node sends a "Hi There" message to the LC)
+ 1 (the LC sends the lock table and up list to the node)
+ NRP (the IC sends an "Accept Lock" or "Accept Release" message to the node for every lock or release lock request pending)

= 5 + NRP

(ii) Lock Controller.

Whenever a node detects a failed LC, it nominates another node to the position of IC.

```
NMR =   1              ( the nominator sends an "Accept
                         Nomination" message to the
                         nominee
      + 1              ( the nominee checks that the old
                         LC is still down )
      + ( NA - 1       ( the nominee notifies every other
                         site that it is the new LC
      + ( NA - 1 )     ( the new LC sends an "Update
                         Table" message to every node )
      + ( NA - 1 )     ( every node sends a "Ready to
                         Update" message to the new LC )
      + ( NA - 1       ( the new LC sends a "Resume
                         Normal Activity" to every other
                         node )
    = 2 + 4 * ( NA - 1
```

- Interrupt for recovery.

  During the time that the nominee is not able to handle all the functions as the new LC, the total system is blocked. Nodes resume work when they receive the message "Resume Normal Activity".

d) STONEBRAKER.

```
NMR =   ( NA -1        ( send a "reconfigure" message
                         to all sites )
      + ( NA - 1 )     ( send an "I'm up" message to
                         all sites
      + ( NA - 1 )     ( send up-list to all sites )
      + ( NA - 1 )     ( every site sends an "I agree"
                         message
      + ( NA - 1 )     ( send a "normal" message to
                         all sites )
    = 5 * ( NA - 1
```

- If another node is detected crashed, a "reconfigure" message is sent to all sites on the up list.

- Interrupt for recovery.

When a site receives the "reconfigure" message, it does not accept more transactions until after it gets the "normal" message, therefore the time it remains blocked is:

TT = (1 + 1 + 1 + 1 + 1) * TT = 5 * TT + K

where K is the time to process the messages.

Nodes concerned = NA - 1

e) CHCU AND LIU.

NMR = 1 or (NA -1 (the node broadcasts an "I-am-up" message to all other sites. The number of messages depends on the communication protocol used
+ 1 (the node asks some other node for its recovery array and the up list)
+ 1 (the node receives the recovery array and the up list)
= 3 or (NA + 1

- Interrupt for recovery.

Any active site which receives the "I-am-up" message updates its up-list, so the interrupt is minimal.

Nodes concerned = 1 (the one sending the recovery array and the up list)

- To detect a crash.

The site detecting the crash creates the recovery array, puts the transaction into it, broadcasts the identification of the crashed site and deletes it from the up-list. Every site will do the same upon being notified of the crash.

5.5. CONCLUSIONS

We have presented a framework for the evaluation of distributed database concurrency control algorithms. This framework includes the most significant parameters which influence the performance of the algorithms. The main performance metrics for concurrency control algorithms are system throughput and transaction response time. Some of the factors that influence these metrics are: intersite communication, load of a node, rejection of transaction and latter restarts, blocking of transactions during normal processing, and interruptions due to recovery procedures. The impact of each of these factors on system throughput and response time varies from algorithm to algorithm, system to system, and application to application. This impact is not understood in detail [BERN81], and an accurate quantitative analysis of performance is difficult to do due to the fact that only very few of these algorithms have been implemented or simulated and statistics are not available yet.

CHAPTER 6

PROGRAMMING DISTRIBUTED ALGORITHMS

6.1. INTRODUCTION

The notion of programming has been evolving in order to adapt to and solve the new problems emerging as a result of advances in computer technology. Initially, sequential programming was developed in order to solve problems in the environment of a single processor and memory. In search of more efficient uses, multiprocessors sharing memory followed by concurrent programming were developed. Real time was developed to solve the problem of controlling real-time events. Advances in hardware technology have led to greatly decreased costs for processors and memory. A possible consequence of the decreased cost is a new way of organizing software, whereby parts of a program reside at and are executed at different computers connected by a network. We will refer to such a program as a distributed program [LISK79]. Recent advances in computer technology, such as computer networks, mini/micro computers and VLSI, have made distributed systems more important and popular [IICM81]. There has been some research done on distributed software, such as distributed operating systems and distributed database systems. To implement these kinds of distributed software, how-

ever, we need some new language concepts that can provide language constructs to handle inherent features in distributed environments. The language features of interest in distributed programming are concurrency, communication, synchronization, time dependency, fault tolerance, and recovery [LICM81,LISK79].

The algorithms for concurrency control in distributed databases described in previous chapters were developed for environments, which present the following characteristics: remote communication, distributed synchronization, problems with node naming, reliability, and time handling. The lack of an adequate tool for explaining these algorithms, has led to difficulty in understanding and implementation them.

Recent research have been oriented towards the development of distributed language/systems and computer networks. Some of these language/systems are DISLANG [LICM81], PLITS [WEID79], and Distributed Processes [BANS78].

6.2. CODING OF THREE ALGORITHMS

In this section we will code three of the previously described algorithms. Due to the lack of a well-known distributed language and the lack of necessary details about the algorithms, pseudo-code will be used to code the algorithms. Once the algorithms are explained using the same methodology, they can be compared more easily. The

algorithms will be compared and conclusions drawn about the complexity of the algorithms and amount of code required. One of the algorithms is a centralized locking one (Menasce et al.), another is a distributed voting one (Thomas), and the third is a distributed one (Ellis . These algorithms are selected because they are familiar and exhibit distinctive characteristics.

6.2.1. Thomas [THOM79]

This algorithm works for fully replicated databases, and for this reason read-only transactions are straightforward. To query the database, an application program (AP) sends a query request to a database managing process (DBMP). The DBMP acts upon the request by querying its copy of the database and returning the results to the requesting AP.

In general, an AP initiates an update by first performing a computation to generate new values for certain database elements using database values obtained by one or more queries, and then submitting an update request to a DBMP which cooperates with the other DBMPs to perform the update.

The skeleton of the update procedure looks like:

```
Procedure :

   QUERY-DATABASE ;
   /* The AP queries the database to
      obtain the values to use in its
```

```
   update computation */

COMPUTE-UPDATE ;
/* The AP computes new values for the
   data elements to be updated */

SUBMIT-REQUEST ;
/* The AP submits an update request
   to a DBMP */

SYNCHRONIZE-UPDATE ;
/* The set of DBMPs cooperates to
   decide to accept or reject the
   request */

APPLY-UPDATE ;
/* If the request is accepted, each
   DBMP applies the update to its
   copy of the database */

NOTIFY-AP ;
/* A DBMP informs the AP how the
   request was resolved */

End.
```

In this section we are concerned with the SYNCHRONIZE-UPDATE step. Update requests made by APs must be communicated among the DBMPs for voting, and DBMP votes must also be communicated to be tallied. This algorithm allows two communication disciplines: Broadcast and Daisy Chain. We will use Daisy Chaining which results in resolution with the minimum number of messages at the expense of relatively long delays.

```
Process SYNCHRONIZE-UPDATE :
/* This process is activated when an update
   request is received by a DBMP */
```

```
Begin

   READ-COPY ;
   /* The DBMP reads its copy of the items
      to update */

   VOTE ;
   /* The DBMP votes on the request */

   CHECK-RESOLUTION ;
   /* After voting, the DBMP checks whether
      its vote resolved the request */

End.



Process VOTE :
/* This process is used by a DBMP to determine
   how to vote on an update request*/

Begin

   COMPARE-TIMESTAMPS ;
   /* Compare timestamps of the request
      variables with the timestamps in the
      local database copy */

   if    any variable is obsolete
   then vote REJECT
   else if    all variables are current and the
              request does not conflict with
              pending requests
        then vote OK
              include the request in the pending list
        else if    all variables are current but
                   the request conflicts with a
                   pending request
              then vote PASS
              else defer voting and remember the
                   request for later consideration

End.



Process CHECK-RESOLUTION :
```

```
/* After voting, a DBMP uses this procedure to
   check whether its vote resolved the request */

Begin

   if    the vote was OK and a majority
         consensus exits
   then  accept the request
         notify all DBMPs that the request
         was accepted
         notify the AP that the request was
         accepted
   else

   if    the vote was REJECT
   then  reject the request
         notify all DBMPs that the request
         was rejected
         notify the AP that the request
         was rejected
   else

   if    the vote was PASS and a majority
         consensus is no longer possible
   then  reject the request
         notify all DBMPs that the request
         was rejected
         notify the AP that the request
         was rejected
   else  forward the request and the votes
         accumulated so far to a DBMP that
         has not voted on it

End.
```

```
Process RESOLUTION :
/* This process is activated when a DFMP is
   notified of the resolution of a request */

Begin

   if   the request was accepted
   then apply the request to the local copy
        reject conflicting requests that were
        deferred because of this request
   else if   the request was rejected
        then vote again to reconsider
             conflicting requests that were
             deferred because of this request

End.
```

6.2.2. Ellis' Decentralized Solution [ELLI77a]

This algorithm works for fully replicated databases. Each node which has a database copy, also has a database controller process which communicates with other database controller processes, and which is the only process updating the database copy at that node.

The controller process may be in three different states. In the passive state the controller is idle and remains waiting for internal requests sent by local users, who want to perform updates. When an internal request is received, the controller progresses to an active state in which checks for conflicts. If no consistency conflicts arise, the controller proceeds to an updating state wherein it coordinates the updating of all copies.

Possible incoming messages to a controller are INT REQ, which denotes internal requests sent by local users, who want to perform updates; EXT REQ, which denotes external requests transmitted by database controller processes at other nodes; and UPD, which denotes update data or functions from controllers at other nodes.

The skeleton the algorithm looks like:

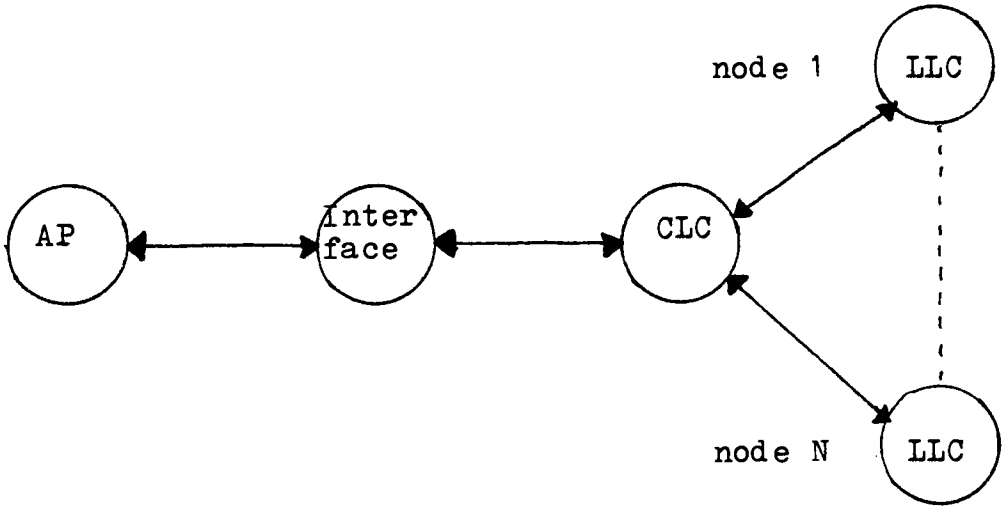
```
Process CONTROLLER:
/* Each node has a copy of this process */

PASSIVE : Wait ( message );
          /* remain in passive state until
             a message arrives */
          case message of
          EXT REQ:
          /* an external request arrives */
             begin
                  send ACK+ to node which sent the EXT REQ;
                  return to passive state;
             end;

          UPD :
          /* an update copy request arrives */

             begin
                  do update in local copy;
                  send ACKd to node which sent the UPD;
                  return to PASSIVE state;

             end;

          INT REQ :
          /* an internal request arrives */
             begin
                  broadcast an EXT REQ to
                  all controllers;
                  go to ACTIVE STATE;
             end;

end;
```

```
ACTIVE: Wait ( message );
        /* remain in active state until the
           request is accepted or rejected */

        case message of

        ACK- :
        /* an external request has been
           rejected by a controller */

           begin
                if ACK- is to an old request
                then return to ACTIVE state
                else send REJECT to user
                     go to PASSIVE state ;
           end;

        UPD :
        /* an update copy request arrives */

           begin
                do update in local copy;
                send ACKd to node which sent the UPD;
                return to ACTIVE state;
           end;

        EXT REQ :
        /* another request arrives, so
           check priorities */

           begin
                if local request has higher priority
                then send ACK- /* reject the
                                  new request*/
                     return to ACTIVE state;

                if local request has lower priority
                then send REJECT to user
                     /* preempt the old request */
                     send ACK+  /* acknowledge the
                                   new request */
                     go to PASSIVE state;

                if priorities are equal /* there is no
                                           chance of
                                           conflict */
                then send ACK+ /* acknowoledge the new
                                  request */
                     return to ACTIVE state;
                end;
```

```
        ACK+ :
        /* an external request has been
           acknowlwdged */

           begin
                if ACK+ is to an old request
                then return to ACTIVE state;
                has the controller received an ACK+
                from every other controller ?
                NO : return to ACTIVE state;
                YES: do update in local copy
                     broadcast an UPD to all
                              controllers
                     go to UPDATING state;
           end;

end;

UPDATING : Wait ( message ) ;
           /* remain in updating state until all
              the controllers have updated their
              copies.  If an EXT REQ arrives, it
              will wait and remain pending until
              the controller returns to passive
              state */

           case message of

           ACKd :
           /* a controller has update its copy
              and sent an ACKd */

              begin
                   has the controller received an
                   ACKd from every other controller ?
                   NO : return to UPDATING state;
                   YES: send DONE to user
                        go to PASSIVE state;
              end;

           UPD :
           /* an update copy request arrives */

              begin
                   do update in local copy
                   send ACKd ;
                   return to UPDATING state;
              end;

end;
```

6.2.3. Menasce et al. [MENA80a]

This algorithms is a locking protocol used to coordinate access to a partially distributed database while maintaining its consistency. Each node on the network has: a central lock controller process (CLC), a local lock controller process (LLC), and an interface process. The interface process receives the lock and release request from the application programs (AP) and sends them to the central lock controller. The central lock controller. The central lock controller is responsible for examining lock and lock release requests from the APs, and deciding whether they should be granted or not. The local lock controller is responsible for keeping a list of all locks in the node, and for checking possible conflicts. The interaction between these processes can be seen in figure 6.1 .



The skeleton of the update procedure looks like :

```
Process INTERFACE :

begin
     loop /* loop forever */
        wait ( event );
        /* remain waiting for messages coming
           from the AP or CLC or timecuts */

        case event of

        message from AP :
           begin
                send request to CLC;
                start timer for request;
           end;

        message from CLC :
           begin
                reset timer for request ;
                send message to AP;
           end;

        time-out :
           begin
                send request to CLC ;
                start timer for request ;

           end;

     endloop ;
end.


Process CENTRAL-CONTROLLER :
/* This process is activated when a lock request
   sent by the interface comes in to the CLC */

   begin
        assign a sequence number to request ;

        CHECK-CONFLICTS :
        /* look at lock table in order to
           determine possible conflicts */

        RELEVANT - NODES :
        /* determine relevant nodes respect
           with the request */
```

```
      BROADCAST;
      /* the lock request is sent to all
         relevant nodes */

      WAIT-ACK ;
      /* wait for ACKs from all relevant nodes */

      CONFIRM-REQUEST ;
      /* a confirmation for the request is
         sent to all relevant nodes */

      UPDATE-TABLES ;
      /* lock tables are updated */

end.
```

```
Process BROADCAST ;
/* This process is used to send a message
   to the relevant nodes */

begin
     for every relevant node do
         send a lock request ;
end.
```

```
Process WAIT-ACK ;
/* This process is activated every time an ACK
   from a relevant node is received */

begin
     if   all relevant nodes have acknowledged
     then if   any of the ACKs was negative
          then send REJECT to interface
               stop ;
end.
```

```
Process LOCAL-CONTROLLER ;
/* This process is used for checking possible
   lock conflicts in each node */

begin
     loop
         wait ( message );
         /* keep waiting for lock requests and
            confirmation messages sent by the CLC */

         If message is a lock request
         then check local table;
              if there is conflict
              then send a negative ACK to CLC
              else send a positive ACK to CLC
              put lock request in pending list ;

         If message is a confirmation
         then delete lock request from pending list;
              put lock request in local lock table;

      endloop;
end.
```

6.3. CONCLUSIONS

Looking at the amount of code that will be necessary to completely code the algorithms, we can conclude that the distributed control algoritms require 1.5 or 2.Ø times more code than the centralized control algorithm. This result agrees with Garcia-Molina's result (GARC78) which compared the simulators used for both types of algorithms.

Since we have used pseudo-code and just touched upon the problem of message sending and retransmissions, the complexity of the algorithms has been diminished. Despite the use of pseudo-code, we can see that distributed control

algorithms are more complex than the centralized control algorithm.

From the previous conclusions, it seems that the distributed control algorithms will probably be harder to implement and more prone to software errors. Of course, this does not mean that the centralized control algorithms should always be chosen over the distributed one. There are many other factors that must be considered before choosing an algorithm such as performance, resiliency, and generality.

CHAPTER 7

CONCLUSIONS

Choosing a concurrency control mechanism for a distributed database is not an easy task. A lot of them have been designed in the past few years, but there are still many unresolved questions about their correctness and performance. Although these mechanisms always solve the same main problem, they use different techniques and offer different services to the user.

In order to choose and implement a concurrency control mechanism in a particular distributed database, we should have detailed knowledge of the particular database, type of transactions and their frequency. expected performance , etc. The available mechanisms can then be compared and one chosen. In order to compare and evaluate the mechanisms, we have proposed criteria which should be fulfilled by the mechanism in general, and some criteria which should be fulfilled depending on the particular application under consideration. We described, evaluated, and code some of the mechanisms.

Further work on distributed concurrency should concentrate on the performance of algorithms. Some ways of achieving this are: a practical simulation of the mechanisms, a

detailed computation of the results, and a quantitative evaluation based on the simulation results.

Finally, we hope that we have fulfilled the objectives of this study, and hope that with the help of future work, a database designer will have the tools to choose the best solutions in terms of user requirements.

BIBLIOGRAPHY

[ADIB81] Adiba M. and Andrade J.M., "Update Consistency and Parallelism in Distributed Databases", Proceedings of 2nd International Conference on Distributed Computing Systems, 1981.

[ALSB76a] Alsberg P. A. and Day J. D., "A Principle for Resilent Sharing of Distributed Resources", Tutorial : Centralized and Distributed Data Base Systems, 1979.

[ALSB76b] Alsberg P. A.,Belford G. G., and Day J. D., "Multi-Copy Resiliency Techniques", Tutorial : Distributed Data Base Management, IEEE 1978.

[BADA78] Badal D, and Popek G., "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Database Systems", Proceedings of the 3rd Berkely Workshop on Distributed Data Management and Computer Networks, 1978.

[BADA80] Badal D., "The Analysis of the Effects of Concurrency Control on Distributed Data Base System Performance", Proceedings of the 6th Conference on Very Large Databases, 1980.

[BAYE80a] Bayer R., Elhardt K., Keller H., and Reiser A., "Distributed Concurrency Control in Database Systems", Proceedings of the 6th Conference on Very

Large Databases, 1980.

[BAYE80b] Bayer R., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems, June 1980.

[BERN78] Bernstein P.A., Rothnie J.B., Goodman N. and Papadimitriou C., "The Concurrency Control Mechanism of SDD-1 : A System for Distributed Databases (The Fully Redundant Case", IEEE Transactions on Software Engineering, May 1978.

[BERN79a] Bernstein P. and Goodman N., "Approaches to Concurrent Control in Distributed Database Systems", Proceedings of AFIPS National Computer Conference, 1979.

[BERN79b] Bernstein P. Shipman D. and Wong W., "Formal Aspects of Serializability on Database Concurrency Control", IEEE Transactions on Software Engineering, May 1979.

[BERN80a] Bernstein P. and Goodman N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", Proceedings of the 6th Conference on Very Large Databases", 1980.

[BERN80b] Bernstein P., Shipman D., and Rothnie J., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980.

[BERN81] Bernstein P. and Goodman N., "Concurrency Control

in Distributed Database Systems", ACM Computing Surveys, June 1981.

[BLCO79] Bloom Toby, "Evaluating Synchrorization Mechanisms", Proceedings of 7th ACM Symposium on Operating Systems Principles, 1979.

[BORR81] Borr Andrea J., "Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing", Proceedings of 7th International Conference on Very Large Databases", 1981.

[CASA81] Casanova Marco Antonic, The Concurrency Control Problem for Database Systems, Lecture Notes on Computer Science 116, Springer-Verlag, 1981.

[CHIL82] Chilenskas R.M., Blaustein Barbara, and Ries Daniel R., "Concurrency After the Fact", Proceedings of Second Symposium on Reliability in Distributed Software and Database Systems, IEEE Computer Society, 1982.

[CHEN80] Cheng, W.K. and Belford,G.G. "Update Synchronization in Distributed Data Bases", Proceedings of the 6th International Conference on Very Large Data Bases, 1980

[CHOU80a] Chou Chuen-Pu and Liu Ming T., "A Concurrency Control Mechanism and Crash Recovery for a Distributed Database System (DLBS)", Distributed Data Bases, North-Holland, 1980.

[CHOU80b] Chou Chuen-Pu and Liu Ming T., "A Concurrency

Control Mechanism for a Partially Duplicated Distributed Database System", Proceedings 1980 Computer Networking Symposium, IEEE 1980.

[DATE81] Date ,C.J. An Introduction to Data Base Systems, Addison-Wesley Publishing Company, Inc. 1981.

[DATE83] Date ,C.J., An Introduction to Data Base Systems , Vol. II, Addison-Wesley Publishing Company, Inc., 1983.

[ELLI77a] Ellis, Clarence A. A Robust Algorithm for Updating Duplicate Databases", Tutorial : Centralized and Distributed Data Base Systems, IEEE Computer Society , 1979.

[ELLI77b] Ellis, Clarence A., "Consistency and Correctness of Duplicated Database Systems", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977.

[DRAK67] Drake Alvin W., Fundamentals of Applied Probability Theory, McGraw Hill Book Company, 1967.

[ESWA76] Eswaran, K.P., Gray ,J.N., Lorie, R.A., and Traiger, I.I., "The Notions of Corsistency and Predicate Locks in a Database System", Communications of the ACM, Nov. 1976.

[GALT82] Galtieri C.A., Gray J., Lindsay P.G. and Traiger I. L., "Transactions and Consistency in Distributed Database Systems", ACM Transactions on Database Systems , Sep. 1982.

[GARC78] Garcia-Molina H. "Performance Comparison of Two Update Algorithms for Distributed Databases", Proceedings of 3rd Berkely Workshop on Distributed Data Management and Computer Networks, 1978.

[GARC82a] Garcia-Molina H. and Wiederhold Gio, "Read-Only Transactions in a Distributed Database", ACM Transactions on Database Systems, June 1982.

[GARC82b] Garcia-Molina H., "Reliability Issues for Fully Replicated Distributed Databases", Computer, IEEE, September 1982.

[GARD80] Gardarin Georges, "Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems", Distributed Databases, North-Holland, 1980.

[GIFF79] Gifford D., "Weighted Voting for Replicated Data", Proceedings of the 7th Symposium on Operating Systems Principles, Sept 1979.

[GRAY76] Gray J.N., Lorie R.A., Putzolu G.R., and Traiger I.L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Modelling in Data Base Management Systems, North-Holland Publishing Company, 1976.

[GRAY78] Gray Jim N., "Notes on Data Base Operating Systems", Operating Systems An Advanced Course, Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

[GRAY81] Gray Jim, "The Transaction Concept : Virtues and Limitations ", Proceedings of the 7th Conference on Very Large Databases, 1981.

[HAMM78] Hammer M. and Shipman D., "An Overview of Reliability Mechanisms for a Distributed Data Base System", Tutorial : Centralized and Distributed Data Base Systems, IEEE Computer Society, 1979.

[HANS75] Hansen, B. "The Programming Language Concurrent PASCAL", IEEE Transactions on Software Engineering, June 1975.

[KAMO81] Kamon F., Kleinrock L. and Muntz R., "Queueing Analysis of the Ordering Issue in a Distributed Database Concurrency Control Mechanism", Proceedings of 2nd International Conference on Distributed Computing Systems, 1981.

[KANE79] Kaneko, A. "Logical Clock Synchronization Method for Duplicated Data Base Control", Proceedings of First International Conference on Distributed Computing Systems, 1979.

[KOHL80] Kohler Walter H., "Overview of Synchronization and Recovery Problems in Distributed Databases", Tutorial - Distributed Processing, IEEE 1980.

[KOHL81] Kohler Walter H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", ACM Computing Surveys, 1981.

[KUNG81] Kung H.T. and Robinson J.T., "On Optimistic Methods

for Concurrency Control", ACM Transactions On Database Systems, June 1981.

[IAMP78] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System" , Communications of the ACM, July 1978.

[LE-L78] Le Lann G., "Algorithms for Distributed Data-Sharing Systems Which Use Tickets", Proceedings of the 3rd Berkely Workshop on Distributed Data Management and Computer Networks, 1978.

[LICM81] Li Chung-Ming and Liu Ming T., "Dislang: A Distributed Programming Language/System", Proceedings of the 2nd International Conference on Distributed Computing Systems, IEEE 1981.

[LIN-81] Lin, Went-Te K., "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Database System", ACM-SIGMOD International Conference on Management of Data, 1981.

[LISK79] Liskov Barbara, "Primitives for Distributed Programming", Proceedings of the Seventh Symposium on Operating Systems Principles, ACM 1979.

[MARTI81] Martin, James. Design and Strategy for Distributed Data processing, Prentice-Hall International Inc., 1981.

[MENA79] Menasce D. and Muntz R., "Locking and Dead-Lock Detection in Distributed Databases", IEEE Transactions on Software Engineering, May 1979.

[MENA80a] Menasce D., Muntz R. and Popek G., "A Locking Protocol for Resource Coordination in Distributed Data Bases", ACM Transactions on Data Base Systems, June 1980.

[MENA80b] Menasce D. and Landes C. F., "On the Design of a Reliable Storage Component for a Distributed Database Management System", Proceedings of the 6th Conference on very Large Databases, 1980.

[MEYE65] Meyer Paul L., Introductory Probability and Statistical Applications, Addison-Wesley Publishing Company, Inc., 1965.

[MOHA81] Mohan C. and Silberschatz A., "Distributed Control - Is It Always Desirable ?", Proceedings of Symposium on Reliability in Distributed Software and Database Systems, July 1981.

[PAPA81] Papadimitriou Christos H., "On The Power of Locking", ACM-SIGMOD International Conference on Management of Data, 1981.

[ROTH80] Rothnie J.B., Bernstein P.A., Fox S., Goodman N., Hammer M., Landers T.A., Reeve C., Shipman D.W. and Wong E., "Introduction to a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980.

[ROSE78] Rosenkrantz D.J., Stearns D.J. and Lewis P.M., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database

Systems, June 1978.

[RYPK79] Rypka D.J. and Lucido A.P., "Deadlock Detection and Avoidance for Shared Logical Resources", IEEE Transactions on Software Engineering, Sept. 1979.

[SEGU79] Seguin J. C., Sergeant G. and Wilms P., "A Majority Consensus Algorithm for the Consistency of Duplicated and Distributed Information", Proceedings of the 1st International Conference on Distributed Computing Systems , 1979.

[SHAP78] Shapiro R.M. and Millstein R.E., "Failure Recovery in a Distributed Data Base System", Tutorial : Centralized and Distributed Data Base Systems, IEEE Computer Society, 1979.

[SKEE82] Skeen Dale, On Network Partitioning , Proceedings of Second Symposium on Reliability in Distributed Software and Database Systems, IEEE Computer Society, 1982.

[STON77] Stonebraker M. and Neuhold Erich, "A Distributed Data Base Version of Ingres", Tutorial : Centralized and Distributed Data Base Systems, IEEE Computer Society, 1979.

[STON79] Stonebraker M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres", IEEE Transactions on Software Engineering, 1979.

[TANE81] Tanenbaum Andrew S., Computer Networks, Prentice-Hall, Inc., 1981.

[THOM7?] Thomas, Robert H. "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases", Tutorial : Centralized and Distributed Data Base Systems, IEEE Computer Society, 1979.

[THOM79] Thomas, Robert H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems, 1979.

[ULLM83] Ullman, Jeffrey D., Principles of Database Systems, Computer Science Press,1983.

[WILM80] Wilms, P. "Qualitative and Quantitative Comparison of Update Algorithms in Distributed Data Bases", Distributed Data Bases, North-Holland, 1980.