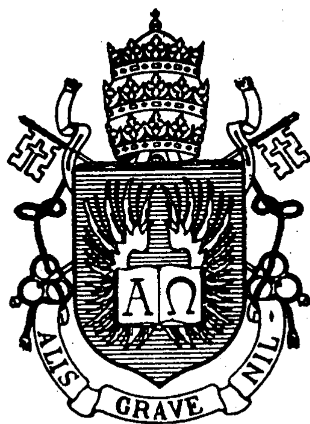


PUC

Series: Monografias em Ciência da Computação

Nº 10/81

CORRECTNESS AND PERFORMANCE EVALUATION OF A TWO-PHASE COMMIT BASED PROTOCOL FOR DDBs

by

Tatuo Nakanishi

Daniel A. Menascé

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

Informática — PUC

DOAÇÃO

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação

Nº 10/81

Editor: Marco A. Casanova September 1981

UC 27546-4

CORRECTNESS AND PERFORMANCE EVALUATION OF A TWO-PHASE COMMIT BASED PROTOCOL FOR DDBs*

by

Tatuo Nakanishi **

Daniel A. Menascé

* This work has been sponsored in part by FINEP

** Present address: Escola Federal de Engenharia de Itajubá
Itajubá, Minas Gerais - Brasil

ABSTRACT:

Many concurrency control algorithms for distributed database management systems have been proposed in the last few years, but little has been done to analyze their performance. This paper presents the specification of a concurrency control algorithm based on the two-phase commit protocol for DDBs. A correctness proof of the algorithm as well as a complete performance analysis is included.

KEYWORDS: Concurrency control algorithms, distributed database management systems, performance analysis, serialization, correctness of concurrency control

RESUMO:

Muitos algoritmos de controle de concorrência para sistemas de gerência de bancos de dados distribuídos foram propostos nos últimos anos, mas pouco se fez para analisar seu desempenho. Este trabalho apresenta a especificação de um algoritmo de controle de concorrência baseado no protocolo de atualização em duas etapas para DDBs. São incluídas uma prova de correção do algoritmo e uma análise completa do seu desempenho.

PALAVRAS CHAVE: algoritmos de controle de concorrência, gerência de bancos de dados distribuídos, análise de desempenho, serialização, correção de controle de concorrência.

1. INTRODUCTION

A database which has its data elements stored at se - veral sites of a computer network is called a distributed data base (DDB). When a DDB is subject to concurrent access from many users submitting their database requests (transactions) at the same or at different sites, the semantic integrity of the database may be violated. Therefore, concurrency control mechanisms have to be incorporated in the design of the distributed DBMS (DDBMS). Many different solutions have been proposed in the last few years in the literature. Just to name a few of them we have: Ellis'Ring Algorithm [1] , Majority Consensus Algorithm [2] , Conflict -Driven Restarts [3], Primary Site Locking [4 and 5] , Primary Copy Locking [6] , SDD-1 Conflict Graph Analysis [7] , Multiversion Algorithm [8], Locking and Timestamps [18], and so on. There are many more concurrency control algorithms which could be added to this list.

A nice and comprehensive study of concurrency control (CC) mechanisms was given by Bernstein and Goodman in [9 and 10]. Their work classify concurrency control mechanisms into timestamp based and locking based ones. An important contribution of the work in [9] is the recognition that CC mechanisms can be obtained by combining a read-write synchronization technique with a write-write synchronization technique. For instance, for timestamp based CC mechanisms they considered three different techniques for read write synchronization and four techniques for write-write synchronization, generating a total of 12 principal CC mechanisms. By introducing some modifications to the basic techniques, over 50 different CC mechanisms can be generated.

In the presence of a such a multitude of CC mechanisms one is faced with the problem of comparing their performance.

The basic motivation of our work is to do a complete study on a concurrency control (CC) algorithm. This study includes a specification, a correctness proof and a detailed performance evaluation. Among the few works on performance of CC mechanisms we have [11, 12, 13, 19 and 14]

Section two of this paper presents very briefly some of the background needed to read this paper. We assume that the reader is familiar with aspects of concurrency control in distributed databases. Section three presents the algorithm in an informal but easy to understand manner. The algorithm uses timestamp ordering to achieve write-write synchronization and uses the two-phase commit protocol to achieve atomic updates. Section four introduces the analytic models used in the analysis, and section five lists the results obtained in the analysis. Section six concludes with the presentation of several curves to illustrate the behavior of the algorithm under several circumstances. Simulation was used to verify the accuracy of the analytic model. A comparison between the simulation and the analytic results also appears in section 6. Appendix A contains a correctness proof of the proposed algorithm and appendix B contains the proofs of all analytic results.

2. BASIC CONCEPTS AND DEFINITIONS

A distributed database (DDB) is a database with its data elements stored at several sites of a computer network.

Certain data items may be redundantly stored at more than one site, for reliability reasons. If a copy of all data items of the database exists at every site then the DB is said to be fully replicated. Otherwise it is said to be partially replicated. In the former case, queries can be completely evaluated at a single site, while in the latter, several query processing alternatives exist. The selection of an optimum query evaluation strategy which minimizes response time has been studied by several authors [15] . We are not interested in studying this effect, but rather the impact of the degree of conflicts between transactions in their response time. Therefore, we assume that the DB is fully replicated since conflicts between transactions will occur irrespective of the degree replication of the DB.

Users interact with the database through transactions which may be of two types: read transactions and update transactions. A read transaction is composed of one or more read actions plus any processing that may be necessary. Since the DDB is fully replicated the read actions can be performed at a single site. An update transaction, besides reading and processing , will update the DB. In this case, all copies must be updated.

The execution of an update transaction can be divided into three phases: a read and processing phase, a pre-commit phase and a commit phase. During the pre-commit phase all information necessary to update the database is stored in stable storage in all sites. This information is called an <u>intentions list</u> [16 and 17] . During the commit phase the database is actually updated through the execution of the intentions list. A detailed explanation of the implementation of intentions lists can be found in [17] .

For each transaction there is a process, called <u>transaction controller (TC)</u>, that manages the execution of the transaction. The TC is assumed to reside at the site where the transaction is introduced in the system (the site of origin of the transaction). Upon arrival at the system, a transaction is assigned a unique timestamp.

3. THE CONCURRENCY CONTROL ALGORITHM

The algorithm presented here is rather simple and is based on well known ideas such as the use of:

i) two-phase commit protocol for atomic implementation of update transactions

ii) locking for conflict detection between transactions

iii) timestamps, delays and restarts for conflict resolution between transactions.

The execution of a read transaction is totally accomplished at the site of origin of the transaction. The read and computation actions of an update transaction are executed at its site of origin while the update actions have to be executed at all sites. During the first phase of the two-phase commit , update information is broadcast to all sites and during the second phase the updates are actually committed to the database.

Locks are used by transactions to indicate their intention to read or update the DB.(A detailed description of the types of locks considered here is given later). In this way, a transaction can be prevented from accessing a database resource if it is locked in an incompatible way by another transaction. Instead of waiting in a queue for the desired resource, transactions retry after a certain delay.

An update transaction must successfully lock all needed resources at its site of origin before reading the database . Locks are held until the end of transaction. During the execution of an update transaction, no other conflicting transaction will be able to lock the resources it needs. These transactions will have to retry some time later.

Once an update transaction locks its resources at its site of origin it must try to lock these resources at the remaining sites.As it is not possible to instantaneously lock the needed resources at all sites, deadlocks may arise. In order to deal with this problem, timestamps are used to establish priorities between transactions. A timestamp is assigned to a transaction by its controller. This timestamp indicates the instant when the transaction succeeded in locking all needed resources. Transactions with smaller timestamps are given higher priority. During the precommit phase an update transaction may be rejected by a transaction of higher priority. The rejected transaction must free all acquired resources and be resubmitted.

An update transaction may start its read actions at a given site even if there are other conflicting read transactions but no other conflicting update transaction at that site. However, in order to start the execution of its update actions an update transaction must wait until all conflicting read transactions finish.

Propagation of lock requests and update information to any site is accomplished in a single message. Let us now describe the different types of messages and lock modes used in our concurrency control algorithm.

The following messages are involved.

a. PRECOMMIT(t) : this message is broadcast from the transction controller of transaction t to all other sites, requesting that they store their intentions lists in stable storage.

b. ACK(t) : this message is sent by a site to the TC of t to indicate that the PRECOMMIT message was accepted and that the intentions list of t was already stored in stable storage.

c. REJECT(t) : this message is broadcast by a site to all other sites to indicate the rejection of the PRECOMMIT of transaction t .

d. COMMIT(t) : this message is broadcast by the transaction controller of t to tell all other sites to execute their intentions lists.

Any database item may be in six different states:

1. U : unlocked

2. (SH,i): locked in share mode by i read transactions. This lock mode is used for read-only transactions.

3. WK : locked in weak mode. An item is put into this state to indicate that an update transaction intends to update it. This type of lock may be lifted by other transactions under certain circumstances as will be seen later.

4. ST : locked in strong mode. This indicates that the item is being update . This type of lock may not be lifted by another transaction.

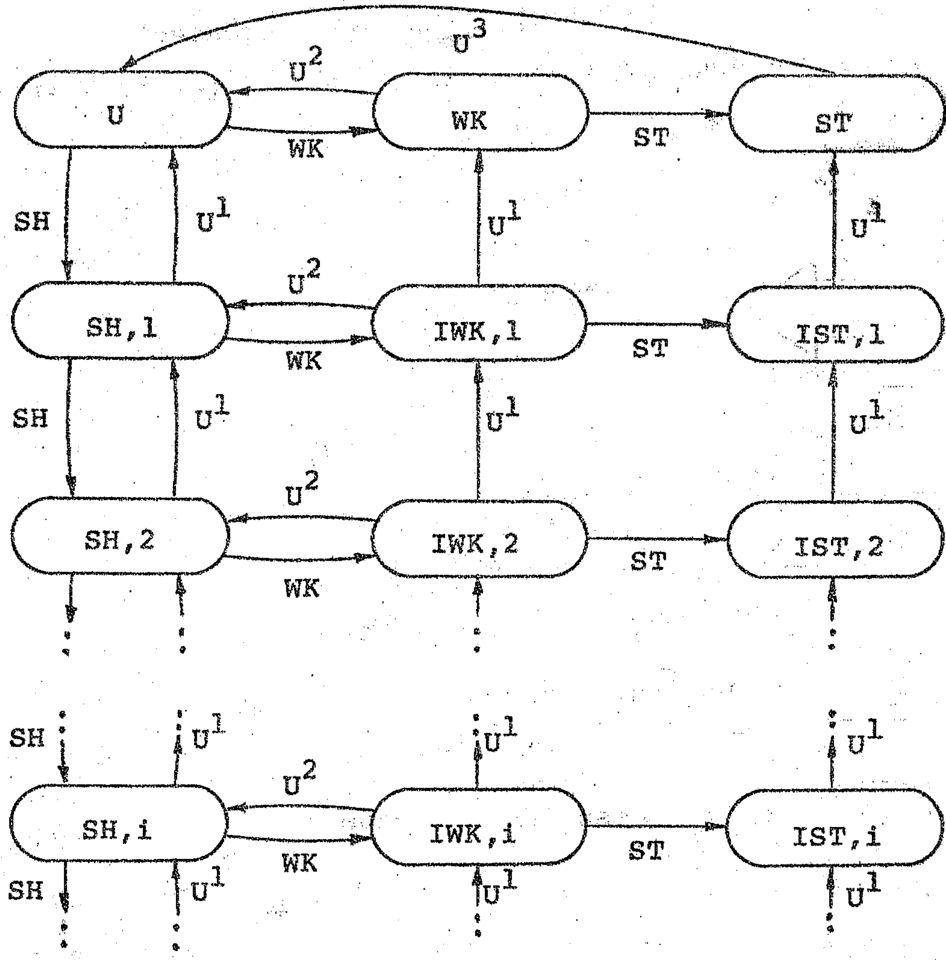
5. (IWK,i) : locked in intention WK mode for one update transaction and locked in SH mode by i(i > 0) read transactions. Whenever a WK lock would be placed on an item locked in SH mode, an IWK lock is used instead. When all read transactions release the SH lock on the item , the IWK lock is converted to WK lock.

6. (IST,i) : locked in intention ST mode by an update transaction and locked in SH mode by i (i > 0) read transactions.

Figure 1 illustrates the possible transitions for the state of a database resource at a given site. The labels on the arrows indicate the kind of lock request which caused the transition. The labels SH, WK, ST and U indicate a request to place the resource in SH, WK, ST and U mode respectively. A superscript is used to distinguish different situations when resources are freed.

An update transaction is only allowed to request to place a resource in ST or IST mode if it is in WK or IWK for the same transaction.

In order to describe the algorithm a graphical representation will be used to help the reader to visualize what is really going on. The following graphical conventions are used:

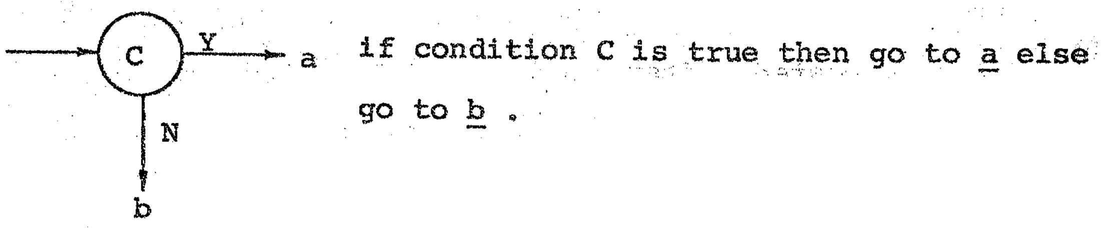


U^1 : the resource is freed by a read transaction.

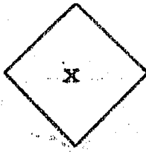
U^2 : the resource is freed by an update transaction rejected during the precommit phase.

U^3 : the resource is freed by an update transaction which completes.

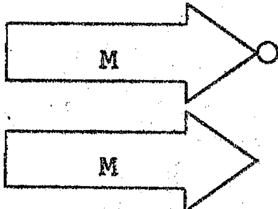
Figure 1 - Transitions Between Lock Modes



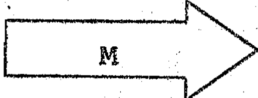
if condition C is true then go to a else go to b .



place all items of a transaction in x mode. The lock mode which results from this request can be found by examining the diagram of figure 1.

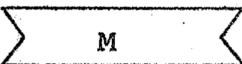


broadcast message M.

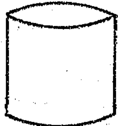


send message M

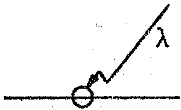
arrival of message M



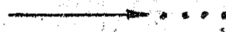
save message M.



acess the database

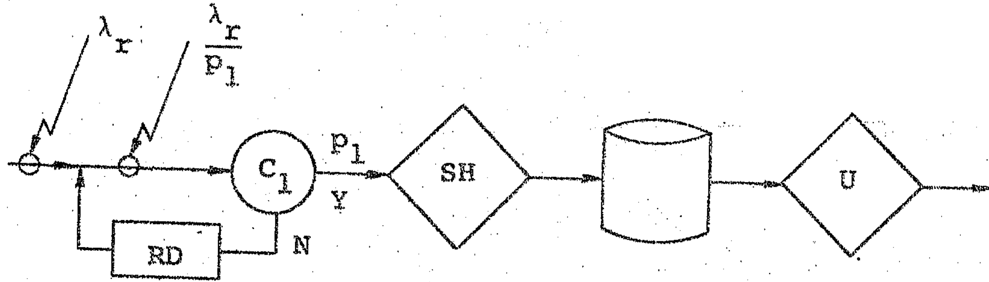


the rate at which transactions cross the circled path is λ



waiting for a message to arrive.

We are now ready to describe the algorithm. The graphical description that follows illustrates the algorithm as executed at a single site. All sites implement the same algorithm. Let us first consider read transactions (see figure 2). When read transactions arrive they perform the test indicated in the figure. If the answer is no (N exit) then the test is repeated after a certain delay. Otherwise, the necessary items are locked in share mode, the database is

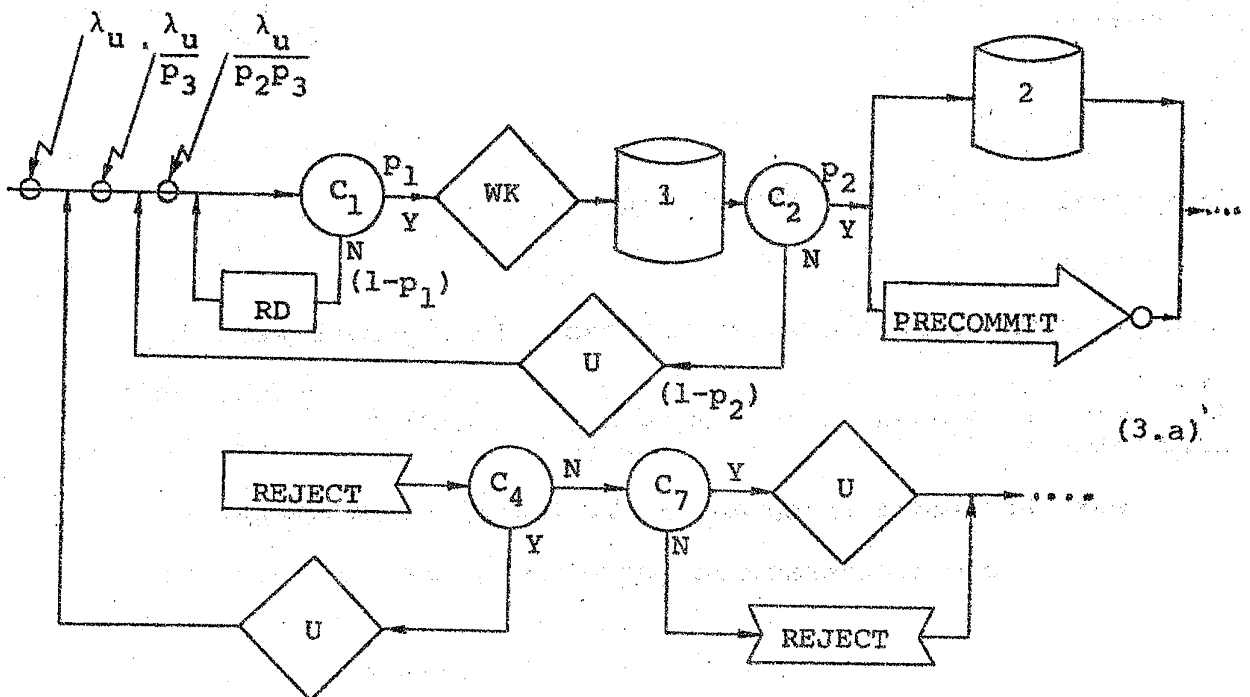


C_1 : are all the resources needed in U or SH mode?

Figure 2 - Read Transaction

read and the resources are unlocked.

The algorithm for update transactions is shown in figure 3.



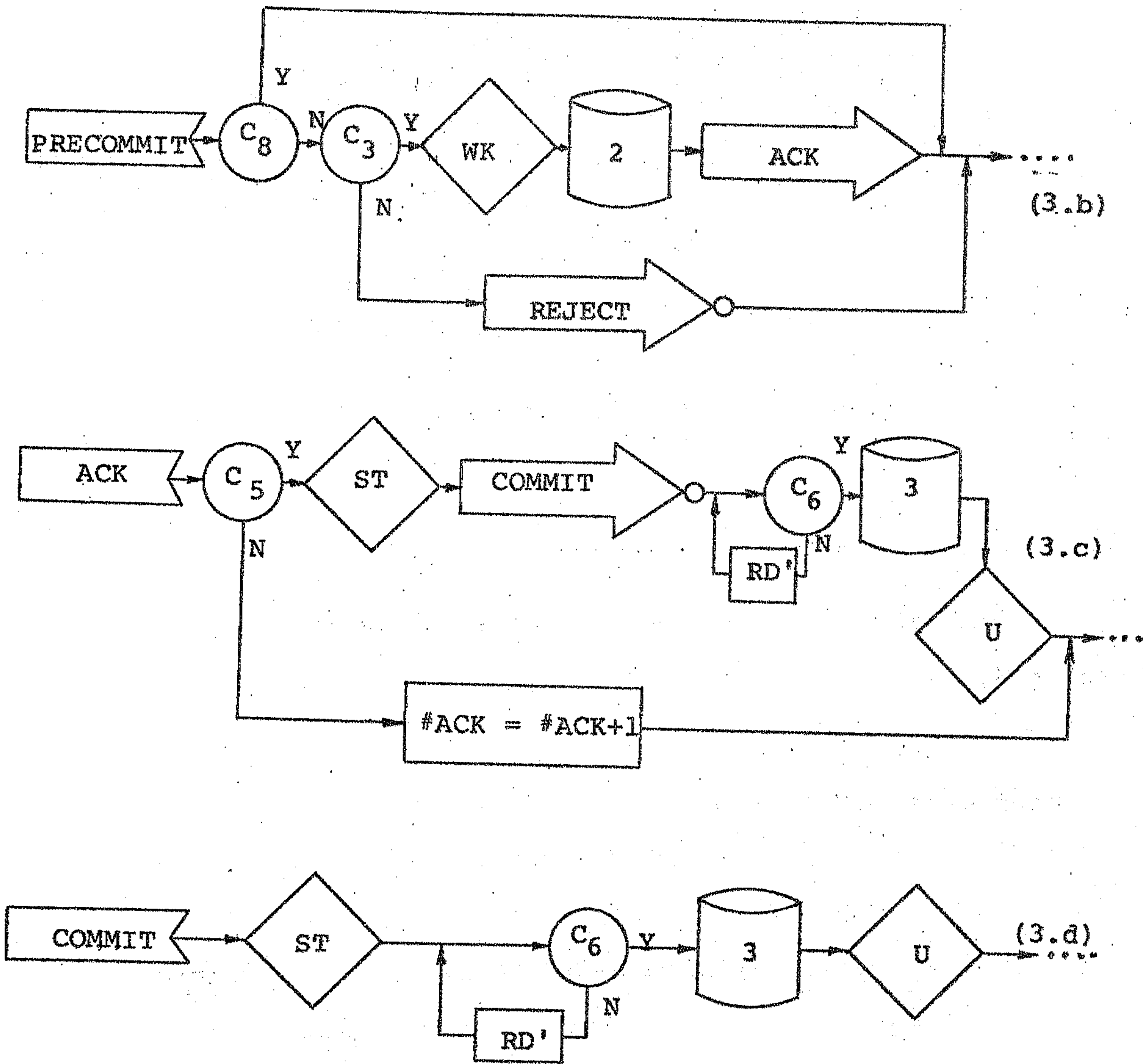


Figure 3 - Update Transaction

The conditions for the tests in figure 3 are

C_1 : are all needed resources in U or SH mode ?

C_2 : are all needed resources still locked in WK or IWK mode for the transaction ?

C_3 : are the needed resources free, locked in SH mode or locked in WK or IWK mode for transactions with greater timestamp?

C_4 : is the node the TC of the transaction?

C_5 : is the number of received ACKs (#ACK) equal to N - 2 ?

C_6 : are all the needed resources in ST mode?

C_7 : has the site already received a PRECOMMIT message for this transaction?

C_8 : is there a saved REJECT for the transaction?

Figure 3.a represents the arrival of an update transaction at a given node. The transaction must initially read the database (see database access box labelled 1), and then, in parallel, start to store its intentions list in stable storage (box labelled 2) and broadcast the PRECOMMIT messages. The arrival of a REJECT message at the site of the TC causes the transaction to be restarted. Figure 3.b shows that when a PRECOMMIT message is received it can be either accepted or rejected. Notice that a PRECOMMIT message may lift locks placed in WK or IWK mode by transactions of greater timestamp. This is the reason why the test of condition C_2 in figure 3.a is necessary. Figure 3.c shows that when all ACKs are received by the TC of the transaction, a COMMIT message is broadcast to all other sites. The database access box labelled 3 represents the execution of the intentions list. Finally, figure 3.d shows what happens when a COMMIT message is received.

As it can be seen from the test of condition C_3, a precommit of a transaction with a smaller timestamp has higher priority over one of greater timestamp. In other words, precommits of conflicting transactions are accepted in timestamp order. This is how write-write synchronization is achieved (see Bernstein 80a). Appendix A contains a proof that any execution generated by this algorithm is serializable, and that read transactions always see consistent data.

Let us comment on the necessity of the test of condition C_6 before actually executing the intentions list of a transaction. Whenever a node is ready to commit, all resources involved are either in ST or IST mode. If all resources are in ST mode then the commit can be done right away. Otherwise, it must be delayed until the IST mode is converted to ST mode. This will happen when read transactions that were reading when the IST mode was installed finish to read. It should be remembered that after the IST mode is granted, no other read transaction is allowed to start. Therefore, it is very likely that no wait will be necessary before executing the commit, since an interval of at least 2T time units (T is the message transmission time) will elapse between the granting of an IST lock and the execution of the commit.

A final comment on the algorithm is that there is no situation in which a transaction has to wait for database resources. This is avoided using delays and retries.

4. MODEL DEFINITION AND PARAMETERS

In this section we present the assumptions and parameters used to construct the model analyzed in this paper.

4.1 Transaction Characterization

The set of resources read by a transaction is called its read set and the set of resources updated by it is called its write set. Our first assumption for transactions is:

T1 : The size of the read set is the same as the size of the write set and is the same (constant) for every transaction.

The next assumption is concerned with the arrival process of transactions

T2 : Read and update transactions arrive at the system according to a Poisson process.The total arrival rate of read or update transactions is equally divided among all sites, i.e. the load is balanced.

The important measure of interest for transactions is their response time. The response time of a transaction is measured as the time interval between the instant the transaction is submitted to the system and the instant when the user is notified of the completion of the transaction. Update transactions are considered to be completed when all sites have written their intentions list into stable storage, since from this point on, the modifications generated by the transaction

will be reflected into the database no matter how many failures occur. For read transactions the completion instant is when all read and processing actions have been executed.

4.2 Network and DDB Assumptions

The assumptions regarding the database are the following:

D1 : The database is fully replicated

D2 : The total number of items which compose the database is constant, i.e. we disregard insertions and deletions to the database.

The assumptions regarding the network are the following:

N1 : The network does not lose nor duplicates messages. Actually, the transport layer at each node implements this property for messages.

N2 : Messages that go from site A to site B arrive at their destination in the same order they were sent.

N3 : The transmission time between a pair of nodes is assumed to be constant.

4.3 Site Assumptions

This section considers the assumptions about the computer system of a site. Our model takes into account that transactions use both CPU and I/O devices and that they compete for the use of these resources. The model used to analyze this effect is shown in figure 4 and is a network of queues.

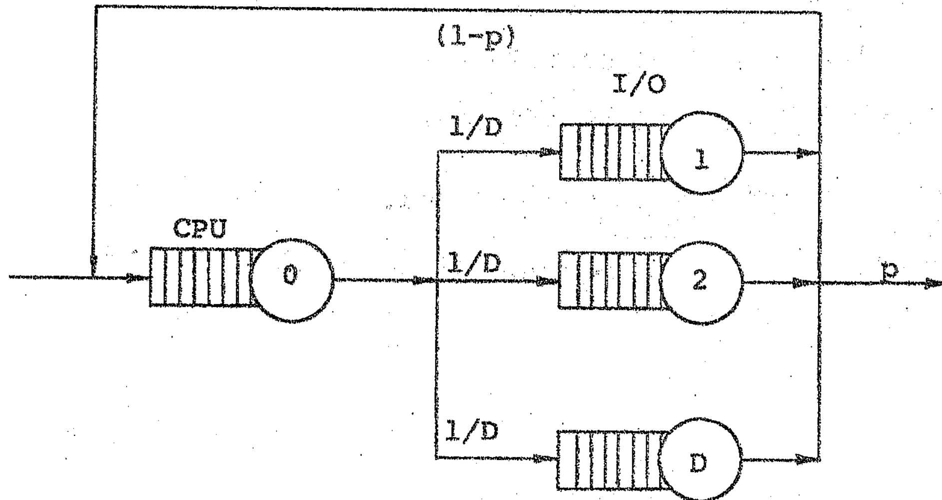


Figure 4 - Model of a Site

As we can see, there are (D + 1) queues. Queue 0 is the CPU queue and the remaining ones are associated with the D existing I/0 devices. After being served at the CPU, transactions move to any one of the I/0 devices with equal probability. The probability p shown in figure 4 was introduced to add another characteristic to transactions, namely their complexity. Complex transactions will have to go through several cycles before they leave the system (small values of p), while simple transactions will require few cycles (p closer to one). The CPU and all I/0 devices are assumed to be exponential severs with FCFS queueing discipline.

4.4 List of Parameters and Performance Measures

A list of all parameters considered in model is given below:

n = number of resources used by a transaction.

λ_r= average arrival rate of read transactions per node.

λ_u= average arrival rate of update transactions per node.

p = probability that a transaction leaves the computer system after being served by an I/O device.

M = number of resources of the database.

D = number of I/O devices per node.

N = number of sites in the network.

T = transmission time of a message between a pair of nodes.

μ_{cpu} = average CPU processing rate.

μ_{io} = average I/O device processing rate

RD = resubmission delay during the read phase.

The performance measures of interest are

R_r = average response time of read transactions.

R_u = average response time of update transactions.

5. PERFORMANCE EVALUATION RESULTS

This section presents the results obtained in our analysis. All derivations appear in Appendix B.

Before we present the results, we would like to comment on some assumptions we made. The first strong assumption used throughout the analysis is that the arrival process of any kind of message is a Poisson process. This approach was taken because otherwise the analytic model would become extremely hard to manage. In order to verify how good an approximation this is, we built a simulation program in SIMSCRIPT II.

Because of space limitations, we do not include the simulation program in this paper, but we would like to add that it was made available to the referees of this paper. Interested readers can obtain a copy of it from the authors. It turned out that the simulation results agreed very well with the analitic ones, as one cann see from table 1.

The other assumption that we would like to comment on is assumption T1, namely that all transactions use a constant number, n, of resources. This choice was adopted because it simplified the analysis and also because we did not have actual data that could serve as an indication of what would be a realistic distribution of the size of the read or write sets. Our assumption implies that the probability of conflict between two transactions, denoted by PC, is

$$PC = \begin{cases} 1 - \frac{\binom{M-n}{n}}{\binom{M}{n}} & \text{if } n \le M/2 \\ 1 & \text{otherwise} \end{cases} \qquad (1)$$

We assume that the average time spent at the computer system to execute the read actions of a transaction and the average time to store its intentions list in stable storage are the same. Let this time be denoted by $\bar{t}$. Then we have the following result.

<u>Result 1:</u>

$$\bar{t} = \frac{1}{\beta}\left(\frac{\rho_{cpu}}{1 - \rho_{cpu}} + \frac{D\rho_{io}}{1 - \rho_{io}}\right) \qquad (2)$$

where $\rho_{cpu} = \frac{\beta}{p\mu_{cpu}}$

$\rho_{io} = \frac{\beta}{pD\mu_{io}}$

$$\beta = \lambda_r + \frac{\lambda_u}{p_3}\left[1 + \frac{1}{p_2} + (N-1)p_3^{\frac{1}{N-1}}\right]$$

and

$p_2 \triangleq P_r$ [an update transaction is not rejected after its read-computation phase but before entering its precommit phase]

$p_3 \triangleq$ [an update transaction is not rejected during its precommit phase]

Result 2: The average response time, R_r, of a read transaction is given by

$$R_r = RD\left(\frac{1-p_1}{p_1}\right) + \bar{t} \qquad (3)$$

where,

$p_1 \triangleq P_r$ [a transaction finds all needed resources free or locked in SH mode at its site of origin]

Result 3: The average response time of an update transaction is given by

$$R_u = \frac{R_r}{p_2 p_3} + \frac{2T}{p_3} + \overline{\max_{N-1}}[\tilde{t}\,] \qquad (4)$$

where the notation $\overline{\max_j}[\tilde{v}]$ indicates the expected value of a random variable defined as the maximum between j random variables distributed as the random variable $\tilde{v}$. Since $\tilde{t}$ is the random variable which indicates the time a node takes to process an external PRECOMMIT message, $\overline{\max_{N-1}}[\tilde{t}\,]$ is the average maximum time it takes the PRECOMMIT message to be processed at the (N-1) nodes.

We derive, in Result 4, an expression for the average of the maximum of a given number of identically distributed random variables. This derivation is rather simple if we assume that $\tilde{t}$ is exponentially distributed with an average $\bar{t}$.

<u>Result 4</u>: Let $\tilde{y}$ be a random variable defined as $\tilde{y} = \max[\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k]$ where $\tilde{x}_1, \ldots, \tilde{x}_k$ are identical and independently distributed random variables. Let the $\tilde{x}_i$ r.v.s be all exponentially distributed with average $\bar{x}$. Then, the expected value of $\tilde{y}$, $\bar{y}$, is given by

$$\bar{y} = \overline{\max_k}[\tilde{x}] = k\,\bar{x} \sum_{j=0}^{k-1} \frac{\binom{k-1}{j}(-1)^j}{(j+1)^2} \qquad (5)$$

<u>Result 5</u>: The probability that a transaction finds all needed resources free or locked in SH mode at its site of origin, p_1, is given by

$$p_1 = \sum_{k=0}^{\lfloor M/n \rfloor} \frac{\binom{M-nk}{n}}{\binom{M}{n}} \frac{\rho^k e^{-\rho}}{k!} \qquad (6)$$

where
$$\rho = \frac{\lambda_u}{p_2 p_3} \{\bar{t} + p_2[2T + p_3 \overline{\max_{N-1}}[\tilde{t}]]\} +$$
$$+ (N-1)\lambda_u p_3^{\frac{2-N}{N-1}} \{T + p_3^{\frac{N-2}{N-1}}[T + \overline{\max_{N-1}}[\tilde{t}]]\}$$

<u>Result 6</u>: The probability, p_2 that an update transaction is not rejected after its read phase but before it enters its precommit phase is given by

$$p_2 = (\gamma\bar{t} + 1)^{1-N} \qquad (7)$$

where $\gamma = \lambda_u\, PC\, p_3^{\frac{2-N}{N-1}}$

Result 7: The probability, p_3, that an update transaction is not rejected during its precommit phase is given by

$$p_3 = \left[\frac{e^{-\alpha T}}{1 - (\alpha \bar{t})^2} \right]^{N-1} \quad (8)$$

where $\alpha = \lambda_u PC \, p_3^{\frac{2-N}{N-1}}$

In order to obtain R_r and R_u, the following procedure should be used.

1. Choose initial values $p_2^{(0)}$ and $p_3^{(0)}$ for the probabilities p_2 and p_3. Recommended values are 0.9999. Set i=0.
2. Compute $\bar{t}$ using $p_2^{(i)}$ and $p_3^{(i)}$ according to Result 1
3. Compute $p_3^{(i+1)}$ using $p_3^{(i)}$ and $\bar{t}$ according to Result 7.
4. Compute $p_2^{(i+1)}$ using $p_3^{(i+1)}$ and $\bar{t}$ according to Result 6.
5. If $\max\{|(p_2^{(i+1)}-p_2^{(i)})/p_2^{(i+1)}|, |(p_3^{(i+1)}- p_3^{(i)})/p_3^{(i+1)}|\}$ is greater than a given tolerance, then set i=i+1 and go to step 2.
6. Compute $\bar{t}$ using $p_2^{(i+1)}$ and $p_3^{(i+1)}$ according to Result 1.
7. Compute p_1 using $p_3^{(i+1)}$, $p_2^{(i+1)}$ and $\bar{t}$ according to Result 5.
8. Compute R_r using p_1 and $\bar{t}$ according to Result 2
9. Compute R_u using R_r, $p_2^{(i+1)}$, $p_3^{(i+1)}$ and $\bar{t}$ according to Result 3.

6. CONCLUSIONS

Many concurrency control algorithms for DDBs have been proposed in the last few years. In the vast majority of cases , their performance analysis has been limited to counting the number of messages or the number of bits transmitted. In this paper we presented a concurrency control algorithm, a correctness proof and a performance evaluation which yielded results such as average response time of read and update transactions and utilization of CPU and I/O devices, as a function of several parameters and as a function of the degree of conflicts between transactions.

The results obtained in the previous section allows us to draw many interesting curves, but due to space limitations, we will concentrate only on four of them.

Figure 5 displays the variation of the average response time of update transactions, R_u, as a function of their average arrival rate. Two sets of curves are shown: one for M = 200 and one for M = 500. For each value of M, two values of $1/\lambda_r$ are considered. As it can be seen from the figure, read transactions have little impact on update transactions. Both sets of curves exhibit a similar behavior. In the beginning R_u grows very modestly since the average arrival rate of transactions is sufficiently small so that conflicting transactions interfere very little with one another. After a certain point, the average arrival rate of update transactions is high enough so that the effect of the degree of conflicts between them starts to impact considerably their average response time.

This effect is superimposed with that of the increased contention for CPU and IO resources in the computer system of each site. The isolation of these two effects can be seen in Figures 6 and 8 discussed below.

Figure 6 shows the interesting effect of the complexity of update transactions, p, on their average response time R_u (p is the probability that a transaction leaves the computer system after being served by an I/O device). A decrease in p represents an increase in the numbers of CPU-IO cycles that will have to be performed by each transaction. In other words for a given arrival rate of update transactions, smaller values of p imply higher contention for computer system resources. As one can see, this increase in load has a heavy impact on R_u. Also indicated in Figure 6 is the average response time, R_u^e, of an update transaction which finds the system empty. These values can be interpreted as a lower bound on R_u, since no interference between transactions is considered.

Each one of the three curves of Figure 8 shows the variation of R_u as a function of the size of the transaction, n, for a given value of $1/\lambda_u$. As the size of a transaction increases, the probability of conflicts between transactions also increases . However, this effect can be better observed for higher values of λ_u. For instance, the curve for $1/\lambda_u$= 3.0 grows much faster than that for $1/\lambda_u$ =20.0. It is worth noticing the dramatic effect that the number of DB resources referenced by a transaction has on performance. Consider the curve for $1/\lambda_u$ = 3.0. For n=10, R_u is 59% higher than its value for n=2, although the number of resources referenced by a transaction grew from 1% to 5% of the total DB resources.

This can be explained by the fact that, for n=2, the probability of conflict, PC, is equal to 0.0199 while for n=10, PC = 0.4085.

Finally Figure 7 shows the variation of the utilization of the CPU and IO devices with λ_u and λ_r.

In order to render our analysis more manageable we made two simplifying assumptions, namely that the arrival process of any type of message at a given site is Poisson and that the computation time at each node is exponentially distributed. A simulation model was developed to verify the validity of these assumptions. As one can see from table 1 the maximum observed error is of the order of 10% even when the system is heavily loaded. In other words, the analytic model is remarkably accurate.

Case	$1/\lambda_r=1/\lambda_u$	R_r sim.	R_r anal.	%Dif.	R_u sim.	R_u anal.	%Dif.
M = 500	10.	.154	.163	-5.8	.689	.718	-4.2
	8.	.156	.167	-7.0	.706	.725	-2.7
	6.	.177	.173	2.3	.739	.737	0.3
	4.	.189	.186	1.6	.774	.761	1.7
	3.	.206	.199	3.4	.807	.788	2.4
	2.	.244	.228	6.6	.869	.845	2.8
	1.5	.280	.262	6.4	.966	.913	5.5
	1.	.377	.348	7.7	1.215	1.087	10.5
M = 200	10.	.173	.176	-1.7	.714	.734	-2.8
	8.	.180	.183	-1.7	.725	.745	-2.8
	6.	.192	.195	-1.6	.767	.764	0.4
	4.	.232	.221	4.7	.843	.805	4.5
	3.	.257	.249	3.1	.895	.850	5.0
	2.	.335	.312	6.9	1.071	.954	10.9

N=6; n=5; D=3; T=0.1; RD=0.5; p=0.2; $1/\mu_{cpu}=0.005$; $1/\mu_{io}=0.025$

Table 1 - Simulation versus Analytic Results.

UNIVERSIDADE CATÓLICA
41950/82

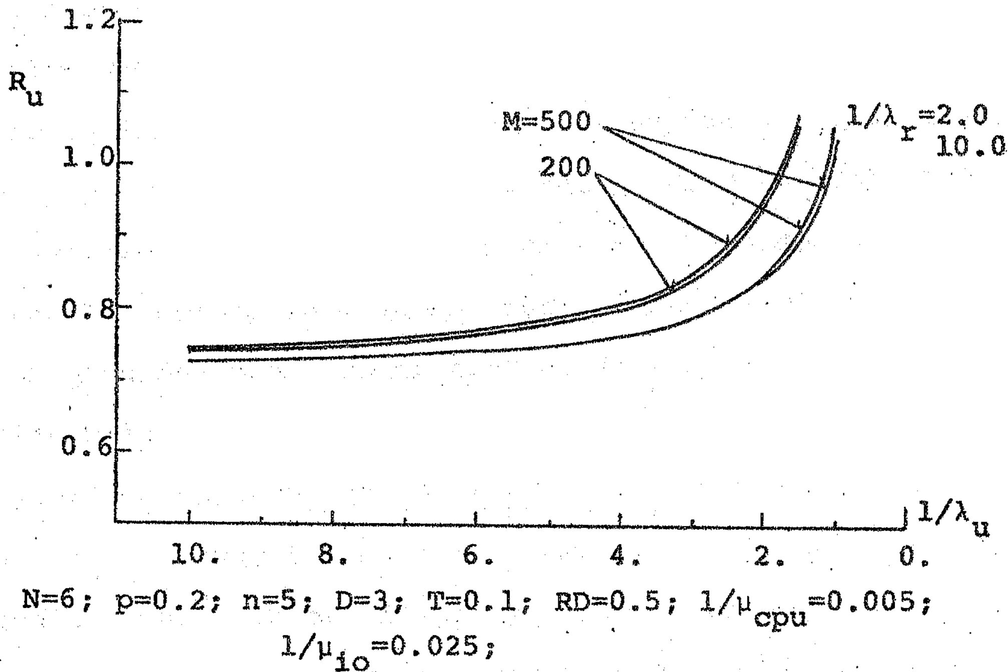


Figure 5 - R_u vs aver.inter arrival time of update trans.

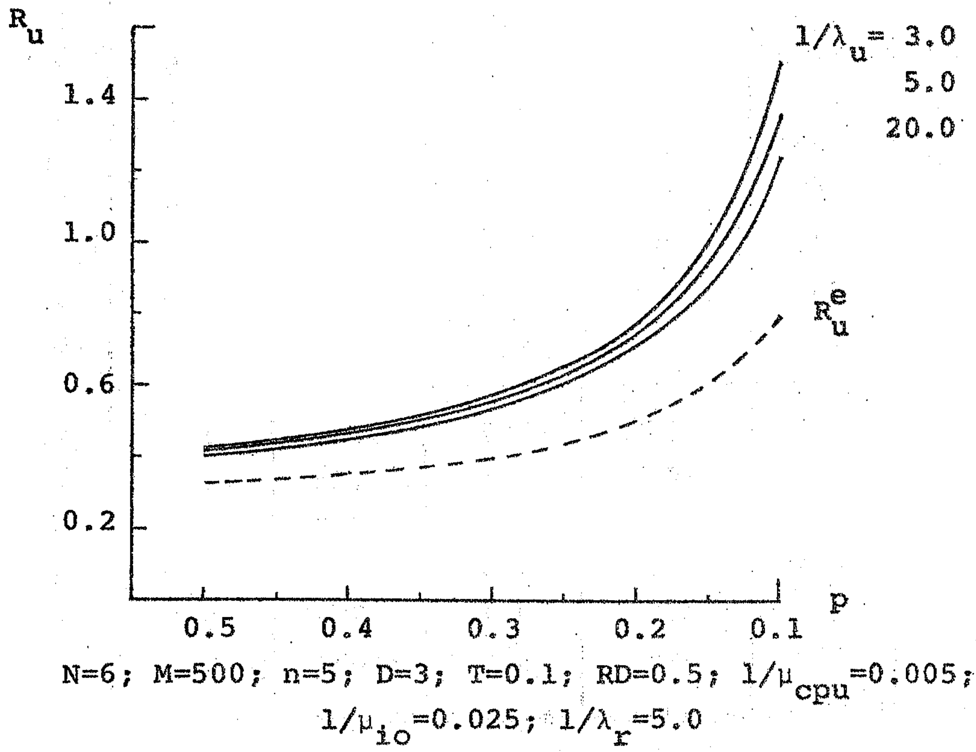
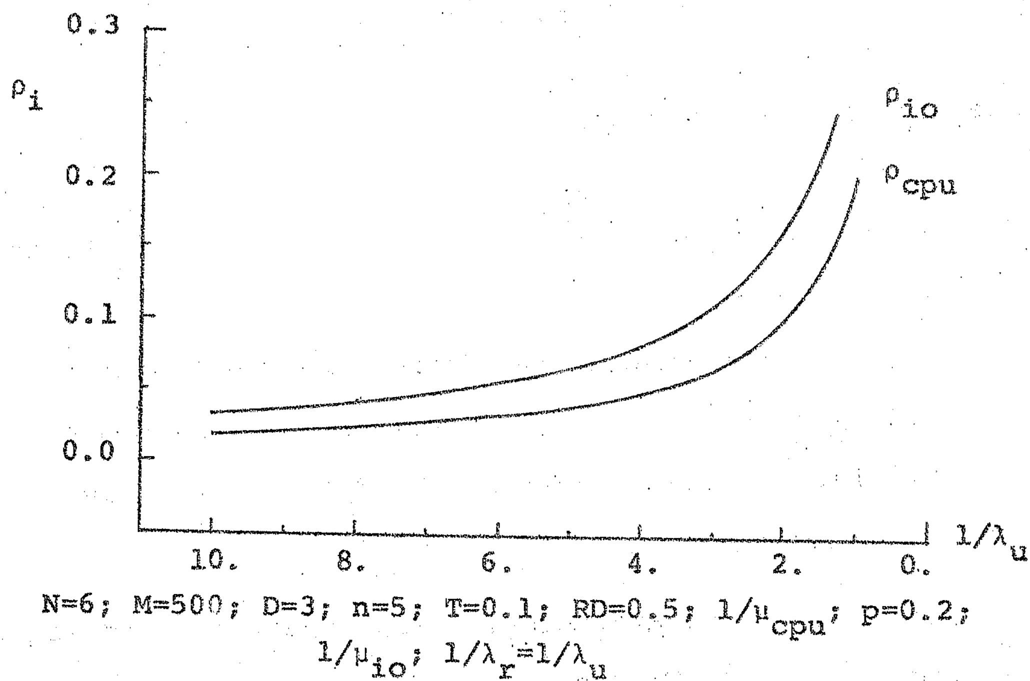
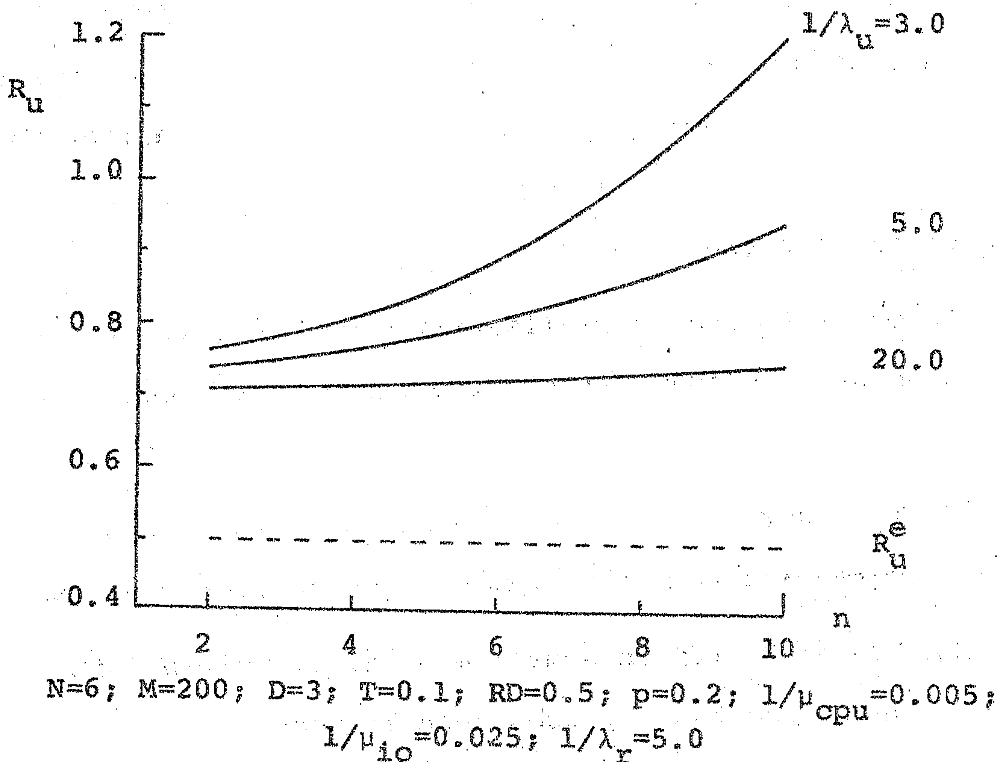


Figure 6 - R_u vs probability p



N=6; M=500; D=3; n=5; T=0.1; RD=0.5; $1/\mu_{cpu}$; p=0.2; $1/\mu_{io}$; $1/\lambda_r = 1/\lambda_u$

Figure 7 - Utilizations vs average inter arrival times



N=6; M=200; D=3; T=0.1; RD=0.5; p=0.2; $1/\mu_{cpu}=0.005$; $1/\mu_{io}=0.025$; $1/\lambda_r=5.0$

Figure 8 - R_u vs number of transaction resources

Appendix A - Correctness Proof of the Algorithm

In order to prove that the algorithm is correct, we are going to show that

i) all read transactions read consistent data

ii) every execution is serializable in the sense that it has the same effect on the database as some serial execution.

The notions of execution, serial execution and log are the same as in [9]. Notice that our notion of serializability differs from the one presented in [9] in the sense that we do not require that read transactions see the same result as they would see in some serial execution, but only that they read consistent data. This is trivially true in our algorithm because read transactions are forced to lock, in SH mode, all resources they need before reading the DB. This prevents conflicting update transactions from writing into the database while read transactions are executing.

In order to show serializability we are going to use Theorem 2 presented in [9]. Let us also repeat here some definitions of relations given in [9]. Let T_i and T_j be transactions.

i) $T_i \rightarrow rw\ T_j$ iff in some log of the execution E, T_i reads some data item into which T_j subsequently writes.

ii) $T_j \rightarrow wr\ T_j$ iff in some log of E, T_i writes into some data item that T_j subsequently reads.

iii) $T_i \rightarrow ww\ T_j$ iff in some log of E, T_i writes into some data item into which T_j subsequently writes.

Let the relation $\rightarrow rwr$ be defined as $\rightarrow rwr = \rightarrow rw \cup \rightarrow wr$

Let us introduce some notation:

$w_i^k[x]$: write action of transaction T_i on data item x at site k.

$r_i^k[x]$: read action of transaction T_i on data item x at site k.

W_i^k : set of all write actions of transaction T_i at site k.

R_i^k : set of all read actions of transaction T_i at site k.

$R_i^k \Rightarrow W_j^k$: any action in R_i^k precedes any action in W_j^k. Similarly for $R_i^k \Rightarrow R_j^k$, $W_i^k \Rightarrow W_j^k$ and $W_i^k \Rightarrow R_j^k$.

$S(T_i)$: site of origin of T_i

log(S): log at site S.

τ_i : timestamp of transaction T_i

Due to our definition of serializability we can completely eliminate read-only transactions in all executions considered in what follows, since these transactions do not alter the state of the database.

The proof that follows is structured into three theorems, six lemmas and some properties. The presentation is top-down in the sense that we start stating and proving our main result which is Theorem 1. In order to prove this theorem we need to prove two other theorems, which need the results of some lemmas. This refinement process is illustrated in figure A.1 which shows how theorems and lemmas depend on each other.

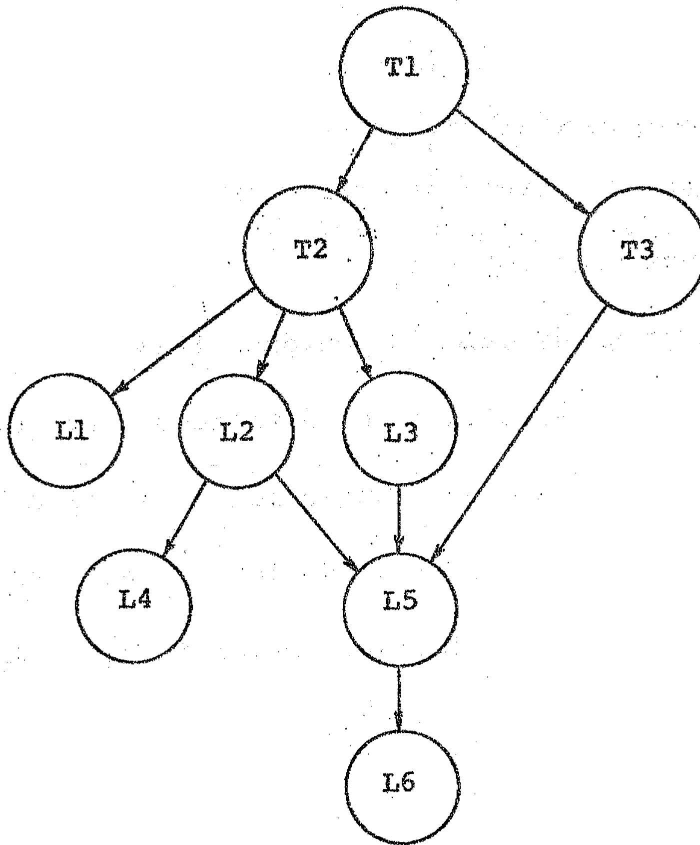


Figure A.1 - Relationship among Theorems (Ti's) and Lemmas (Li's).

Not included in figure A.1 are some properties of the algorithm which are used in the proof of some theorems and lemmas. These properties can be directly derived from the specification of the algorithm given in section 3, and are listed after all theorems and lemmas.

Let E be an execution of the transactions modeled by the set of all logs at every site.

Theorem 1: Any execution E generated by the algorithm given here is serializable.

Proof: According to Theorem 2 in [9], E is serializable if $\rightarrow = \rightarrow rwr \cup \rightarrow ww$ is acyclic.

Assume that $\rightarrow$ possesses a cycle given by

$$T_1 \rightarrow T_2,\ T_2 \rightarrow T_3, \ldots,\ T_k \rightarrow T_1$$

By Theorems 2 and 3 and by the definition of $\rightarrow$, it follows that

$$\tau_1 < \tau_2 \ \&\ \tau_2 < \tau_3 \ldots \ \&\ \tau_k < \tau_1$$

Contradiction. Then, $\rightarrow$ is acyclic,

Q.E.D.

<u>Theorem 2</u>: Let T_i and T_j be two conflicting update transactions. If $T_i \rightarrow rwr\ T_j$ is a pair in the relation $\rightarrow rwr$ associated to the execution E, then $\tau_i < \tau_j$.

<u>Proof</u>: Let us consider the two possible cases:

Case 1: $T_i \rightarrow rw\ T_j$

By definition of the $\rightarrow rw$ relation, there is a site k such that in log(k) there is a data item x such that $r_i^k[x]$ precedes $w_j^k[x]$. From Lemma 1, it follows that $R_i^k \Rightarrow W_j^k$, and from lemma 2 we have that $\tau_i < \tau_j$.

Case 2: $T_i \rightarrow wr\ T_j$

By definition of the $\rightarrow wr$ relation, there is a site k such that in log(k) there is a data item x such that $w_i^k[x]$ precedes $r_j^k[x]$. From Lemma 1, we have that $W_i^k \Rightarrow R_j^k$, and from lemma 3 it follows that $\tau_i < \tau_j$

Q.E.D

<u>Theorem 3</u>: Let T_i and T_j be two conflicting update transactions. If $T_i \rightarrow ww\ T_j$ is a pair in the relation $\rightarrow ww$ associated to the execution E, then $\tau_i < \tau_j$.

Proof: By the definition of $\rightarrow ww$, if $T_i \rightarrow ww\ T_j$ then in some $log(S)$ of the execution E, $w_i^S[x]$ precedes $w_j^S[x]$ for some data item x. According to Property 7, $W_i^S \Rightarrow W_j^S$. From Lemma 5 it follows that $\tau_i < \tau_j$.

Q.E.D.

Lemma 1: Let T_i and T_j be two conflicting update transactions in an execution E. In $log(S(T_i))$ either $R_i^{S(T_i)} \Rightarrow W_j^{S(T_i)}$ or $W_j^{S(T_i)} \Rightarrow R_i^{S(T_i)}$.

Proof: Let us start the proof of this lemma by noticing that since T_i and T_j are transactions in our execution E, then neither one of them is rejected.

If T_i arrives at $S(T_i)$ after T_j succeeded in locking all its resources, then T_i will only be able to start reading after T_j terminates. Therefore, all write actions in $W_j^{S(T_i)}$ will precede all read actions in $R_i^{S(T_i)}$ (i.e., $W_j^{S(T_i)} \Rightarrow R_i^{S(T_i)}$, as indicated by properties 2 and 3 below.

Let us now consider the case in which T_i arrives before T_j locks its resources in $S(T_i)$. If the PRECOMMIT message of T_j arrived at $S(T_i)$ while T_i is in execution then either T_i or T_j would be rejected (depending on their timestamps). Since this is not the case, the PRECOMMIT message of T_j must arrive at $S(T_i)$ after T_i finishes. Therefore, all read actions of T_i will precede all write actions of T_j at $S(T_i)$. In other words, $R_i^{S(T_i)} \Rightarrow W_j^{S(T_i)}$.

Q.E.D.

It is worth observing that Lemma 1 and the following ones depend on the fact that the DB is fully replicated otherwise one could not say that T_j updates data stored at $S(T_i)$.

<u>Lemma 2</u>: Let T_i and T_j be two conflicting update transactions in an execution E. If in the log of a site, S, $R_i^S \Rightarrow W_j^S$ then $\tau_i < \tau_j$.

<u>Proof</u>: By property 7, either $W_i^S \Rightarrow W_j^S$ or $W_j^S \Rightarrow W_i^S$. By property 2, $R_i^S \Rightarrow W_i^S$ and by assumption $R_i^S \Rightarrow W_j^S$. Then, as a consequence of lemma 4, $W_i^S \Rightarrow W_j^S$. Then, from lemma 5 it follows that $\tau_i < \tau_j$.

Q.E.D.

<u>Lemma 3</u>: Let T_i and T_j be two conflicting update transactions in an execution E. If in the log of some site S, $W_i^S \Rightarrow R_j^S$ then $\tau_i < \tau_j$.

<u>Proof</u>: By property 2, $R_j^S \Rightarrow W_j^S$. Since $W_i^S \Rightarrow R_j^S$, then $W_i^S \Rightarrow W_j^S$. By lemma 5, it follows that $\tau_i < \tau_j$.

Q.E.D.

<u>Lemma 4</u>: Let T_i and T_j be two conflicting update transactions in an execution E. At $S(T_i)$, no action of $R_j^{S(T_i)}$ or $W_j^{S(T_i)}$ may appear between $R_i^{S(T_i)}$ and $W_i^{S(T_i)}$.

<u>Proof</u>: First consider that $S(T_i) = S(T_j)$. Then, by properties 2 and 3, T_j cannot start its execution while T_i is executing. Since $R_j^{S(T_i)}$ must precede $W_j^{S(T_i)}$ we cannot have either $R_j^{S(T_i)}$ nor $W_j^{S(T_i)}$ between $R_i^{S(T_i)}$ and $W_i^{S(T_i)}$. Now, let us consider the case where $S(T_i) \neq S(T_j)$.

Since neither T_i nor T_j are rejected given that both of them appear in the execution, the PRECOMMIT message of T_j must arrive at $S(T_i)$ either before T_i starts its execution or after it completes. In the former case, T_i started the execution of $R_i^{S(T_i)}$ after T_j completes property 3, and in the latter case all actions in $W_j^{S(T_i)}$ follow all write actions in $W_i^{S(T_i)}$. In either case, the lemma is proved.

Q.E.D.

<u>Lemma 5</u>: Let T_i and T_j be two conflicting update transactions in an execution E. If in the log of any site S, $W_i^S \Rightarrow W_j^S$ then $\tau_i < \tau_j$.

<u>Proof</u>: By property 1, either $\tau_i < \tau_j$ or $\tau_j > \tau_i$ for $i \neq j$. Let us assume that $\tau_i > \tau_j$ and show that this leads to a contradiction. By property 1, if $\tau_i > \tau_j$ then T_i will start to execute after T_j starts its execution. By property 3, T_i will only be able to start to execute in two cases:

i) before T_j locks its resources at $S(T_i)$

ii) after T_j unlocks its resources at $S(T_i)$

In case i), according to lemma 6, T_i will be rejected, which is impossible since T_i appears in the execution E. In case ii) any write action of T_i will occur at any site after T_j commists at every site (properties 4 and 5). Therefore, $W_j^S \Rightarrow W_i^S$ for every site S which contradicts the assumption that $W_i^S \Rightarrow W_j^S$ for some site S.

Q.E.D.

Lemma 6: Let T_i and T_j be two conflicting update transactions. Let $S(T_i) \neq S(T_j)$. If T_i starts its execution at $S(T_i)$ after T_j starts its execution at $S(T_j)$ and before the PRECOMMIT of T_j arrives at $S(T_i)$, then T_i will be rejected.

Proof: There are two cases to consider:

i) the PRECOMMIT message of T_j arrives at $S(T_i)$ before T_i is able to send its PRECOMMIT messages.

ii) the PRECOMMIT message of T_j arrives at $S(T_i)$ after T_i sends its PRECOMMIT messages.

In case i), T_i will be rejected at $S(T_i)$ and resubmitted, according to properties 2 and 5. In case ii), according to properties 2 and 6, T_i will be rejected at $S(T_j)$. In any case T_i will be rejected and the lemma is proved.

Q.E.D.

Let us now list the relevant properties of the algorithm as mentioned above.

Property 1: A transaction is said to start its execution when it acquires all resources it needs. At this point, the transactions is assigned a globally unique timestamp, which indicates the instant of time at which the execution is being started. So, if T_i starts to execute before T_j, then $\tau_i < \tau_j$.

Property 2: Every read transaction T_i may only start to execute at $S(T_i)$ after it locks all needed resources in SH mode.

Every update transaction T_j will only start to read the database at $S(T_j)$ after it locks all needed resources in WK or IWK mode at that node.

Property 3: If an update transaction is locking resources in WK, IWK, ST or IST mode at a site S, no other conflicting (read or update) transaction will be able to start its execution at site S (see test C_1 in figures 2 and 3).

Property 4: The precommit phase of an update transaction T_j starts after all read actions of T_j have been executed at $S(T_j)$. The commit phase of T_j starts after all resources needed by it are locked in WK or IWK mode at all sites.

Property 5: If the PRECOMMIT message of a transaction T_j arrives at $S(T_i)$ during the read and processing phase of an update transaction T_i, and if $\tau_j < \tau_i$ then T_i will be rejected and re-submitted at the end of this phase. (see test C_2 in figure 3).

Property 6: When a PRECOMMIT message associated with transaction T_j arrives at site S, an ACK message will be sent to $S(T_j)$ if none of the resources needed by T_j are locked at site S by a conflicting transaction T_k such that $\tau_k < \tau_j$. Otherwise the PRECOMMIT is rejected and a REJECT message is broadcast to all sites. If at any site S a REJECT message for a transaction is received before its corresponding PRECOMMIT message then, the latter is ignored when it arrives.

<u>Property 7</u>: Let T_i and T_j be two conflicting update transactions in an execution E. At the log of any site S, either $W_i^S \Rightarrow W_j^S$ or $W_j^S \Rightarrow W_i^S$. This is true since during the execution of all write actions of an update transaction all its resources are locked in ST mode.

Appendix B - Proofs of the Analytic Results

Proof of Result 1: The proof of this result is based on the theory of queueing networks (see [20]). Consider the queueing network model of figure 4. If we let β be the external arrival rate of processing requests to the computer system and λ_i be the total arrival at device i, we have that

$$\lambda_0 = \beta + (1-p) \sum_{i=1}^{D} \lambda_i$$

and (B.1)

$$\lambda_i = \frac{\lambda_0}{D} \qquad \text{for } 1 \le i \le D$$

Then, $\lambda_0 = \frac{\beta}{p}$ and $\lambda_i = \frac{\beta}{pD}$ for $i = 1, \ldots, D$.

According to Jackson's Theorem, [21] , the equilibrium probability, $p(k_0, k_1, \ldots, k_D)$ of finding k_o requests at the CPU and k_i requests at I/o device numbered i, is given by

$$P(k_0, k_1, \ldots, k_D) = P_0(k_0) P_1(k_1) \ldots P_D(k_D) \qquad \text{(B.2)}$$

where

$$P_i(k_i) = (1 - \rho_i)\rho_i^{k_i} \qquad \text{for } i = 0, \ldots, D$$

and

$$\rho_0 = \rho_{cpu} = \frac{\lambda_0}{\mu_{cpu}} = \frac{\beta}{p\mu_{cpu}}$$

$$\rho_i = \rho_{io} = \frac{\lambda_i}{\mu_{io}} = \frac{\beta}{pD\mu_{io}} \qquad \text{for } i = 1, \ldots, D$$

Then, the average number of requests, N_i, at server i is given by

$$N_i = \frac{\rho_i}{1 - \rho_i} \qquad \text{(B.3)}$$

The average total number of requests in the computer system is given by $\sum_{i=0}^{D} N_i$ and from Little's Result the average time, $\overline{t}$, in the computer system is given by

$$\overline{t} = \frac{1}{\beta} \sum_{i=0}^{D} \frac{\rho_i}{1 - \rho_i} = \frac{1}{\beta} \left(\frac{\rho_{cpu}}{1 - \rho_{cpu}} + \frac{D\rho_{io}}{1 - \rho_{io}}\right) \qquad (B.4)$$

The average total external arrival rate β is the sum of arrival rates of the following types of requests:

- read and processing requests by read transactions: λ_r
- read and processing requests by update transac - tions: $\frac{\lambda_u}{p_2 p_3}$
- requests to store the intentions list at the site of origin: $\frac{\lambda_u}{p_3}$
- requests to store the intentions list due to transactions originated remotely: $(N - 1)\lambda_u p_3^{\frac{2-N}{N-1}}$

Hence, β is given by the expression

$$\beta = \lambda_r + \frac{\lambda_u}{p_3}[1 + \frac{1}{p_2} + (N - 1)\ p_3^{\frac{1}{N-1}}] \qquad (B.5)$$

<u>Proof of Result 2:</u> A read transaction succeeds in locking the needed resources in share mode with probability p_1 and fails with probability $(1 - p_1)$. Each time it fails it waits for RD time units before trying again. So, the probability of a read transaction being submitted <u>i</u> times is $(1 - p_1)^i p_1$. The <u>i</u> unsuccesful trials take iRD units of time and the transaction takes $\overline{t}$ units of time to read the database when it succeeds. Hence,

$$R_r = \sum_{i=0}^{\infty} iRD(1 - p_1)^i p_1 + \overline{t} =$$

$$= RD\left(\frac{1 - p_1}{p_1}\right) + \overline{t} \qquad (B.6)$$

<u>Proof of Result 3:</u> Let t_1 be the average time spent by an update transaction since it enters its site of origin until it is ready to send the PRECOMMIT messages. This time is equal to iR_r with probability $(1 - p_2)^{i-1}p_2$ when the read phase is repeated $\underline{i}$ times. Therefore,

$$t_1 = \sum_{i=1}^{\infty} iR_r(1 - p_2)^{i-1}p_2 =$$

$$= \frac{R_r}{p_2} \qquad \text{(B.7)}$$

Now, if the transaction is rejected during the precommit phase, with probability $(1 - p_3)$, its average response time will be equal to $t_1 + 2T$. The term 2T accounts for the time for the PRECOMMIT message to reach any node and for a REJECT message to return to the site of origin of the transaction. Let us now consider the case where the transaction is accepted with probability p_3. T units of time are spent to convey a PRECOMMIT message to all sites. Before sending an ACK message each node must store the intentions list. This operation takes $\bar{t}$ units of time. The transaction controller must wait until all (N-1) ACKs from other nodes arrive. Then, on the average a node must wait $T + \overline{\max_{N-1}}[\bar{t}]$ units of time to receive all ACKs after it sent all PRECOMMITS. If the read and precommit phases are executed j times, the conditioned average response time is given by

$$R_u|j = j(t_1 + 2T) + \overline{\max_{N-1}}[\bar{t}] \qquad \text{(B.8)}$$

Unconditioning on j we get the final expression for R_u as follows

$$R_u = \sum_{j=1}^{\infty} [j(t_1 + 2T) + \overline{\max_{N-1}}[\bar{t}]](1 - p_3)^{j-1}p_3 =$$

$$= \frac{R_r}{p_2 p_3} + \frac{2T}{p_3} \max_{N-1}[\tilde{\varepsilon}] \quad (B.9)$$

Proof of Result 4: Let $F_{\tilde{y}}(y) = p[\tilde{y} \le y]$ be the P.D.F. of $\tilde{y}$. Then

$$F_{\tilde{y}}(y) \triangleq p[\tilde{y} \le y] = \{p[\tilde{x} \le y]\}^k = [\, F_{\tilde{x}}(y)\,]^k \quad (B.10)$$

Since all k random variable are identical and independently distributed. Now, the p.d.f of $\tilde{y}$ is

$$f_{\tilde{y}}(y) = \frac{d}{dy} F_{\tilde{y}}(y) = k\, [\, F_{\tilde{x}}(y)\,]^{k-1} f_{\tilde{x}}(y) \quad (B.11)$$

Since $\tilde{x}$ is exponentially distributed we know that

$$F_{\tilde{x}}(y) = 1 - e^{-y/\overline{x}} \quad \text{and} \quad f_{\tilde{x}}(y) = \frac{1}{\overline{x}} e^{-y/\overline{x}}$$

then,

$$f_{\tilde{y}}(y) = \frac{k}{\overline{x}} (1 - e^{-y/\overline{x}})^{k-1} e^{-y/\overline{x}} .$$

If we let $y^*(s)$ be the L.T, (Laplace Transform) of $f_{\tilde{y}}(y)$ we get

$$Y^*(s) = \frac{k}{\overline{x}}\, G^*(s + 1/\overline{x}) \quad (B.12)$$

where $G^*(s)$ is the L.T. of $(1 - e^{-y/\overline{x}})^{k-1}$. But since

$$(1 - e^{-y/\overline{x}})^{k-1} = \sum_{j=0}^{k-1} \binom{k-1}{j} (-1)^j e^{-jy/\overline{x}} \quad (B.13)$$

it follows that

$$G^*(s) = \sum_{j=0}^{k-1} \frac{\binom{k-1}{j}(-1)^j}{s + j/\overline{x}} \quad (B.14)$$

Now, using (B.14) in (B.12) we get

$$Y^*(s) = \frac{k}{\overline{x}} \sum_{j=0}^{k-1} \frac{\binom{k-1}{j}(-1)^j}{s + (j+1)/\overline{x}} \quad (B.15)$$

Using the moment generating properties of the L.T. we have that

$$\overline{y} = -\frac{d}{ds} Y^*(s)\Big|_{s=0} = k\overline{x} \sum_{j=0}^{k-1} \frac{\binom{k-1}{j}(-1)^j}{(j+1)^2} \qquad \text{(B.16)}$$

<u>Proof of Result 5</u>: Before a transaction is able to start its read and computation phase, it must lock its resources in SH mode. It will only be successful if these resources are not locked by an update transaction. Let T_i be a (read or update) transaction entering at site i.

In order to study the locking process we are going to consider an M/G/∞ model in which each database resource is modeled as a server. Although the number of these servers is finite, the arrival stream to our M/G/∞ system will be composed exclusively by transactions which succeed in locking resources. This implies that these transactions do not have to wait for resources and that transactions in the input stream lock mutually exclusive resources. An infinite server model is thus appropriate. We assume that the arrival process is Poisson. This is an approximation but simulation results indicate that it is a good one (see section 6). We do not make any assumption regarding the distribution of the time a resource remains locked.

Then, the probability P(k) that there are k transactions in the M/G/∞ is given by [22].

$$P(k) = \frac{(\lambda\overline{x})^k e^{-\lambda\overline{x}}}{k!} \qquad \text{(B.17)}$$

where λ= arrival rate at a node of update transactions which succeed in locking resources

$\overline{x}$= average time during which resources remain locked.

Then, if there are k update transactions locking resources, the number of locked resources is nk. The probability that transaction T_i succeeds to lock its resources at site i, given that there are k transactions in the system is given by

$$p_1|k = \frac{\binom{M-nk}{n}}{\binom{M}{n}} \tag{B.18}$$

Unconditioning on k and using (B.17) we get

$$p_1 = \sum_{k=0}^{\lfloor M/n \rfloor} \frac{\binom{M-nk}{n}}{\binom{M}{n}} \frac{(\lambda\bar{x})^k e^{-\lambda\bar{x}}}{k!} \tag{B.19}$$

Let us now calculate λ and $\bar{x}$.

λ_1 = average arrival rate of update transactions which enter the system at site i and manage to lock its resources

$\bar{x}_1$ = average time during which λ_1 type transactions keep their resources locked.

λ_2 = average arrival rate of transactions which enter at other sites and succeed to lock their resources at site i.

$\bar{x}_2$ = average time during which λ_2 type transactions keep their resources locked.

A λ_1 type transaction spends $\bar{t}$ units of time at site i to perform its read and computation phase. With probability $(1-p_2)$ it will be rejected and its resources will be freed, and with probability p_2 it will keep its resources locked during the precommit phase. If the precommit phase is succesful (with probability p_3) it will last for $2T + \overline{\max_{N-1}}[\bar{t}]$ units of time, while if it is unsuccesful it will last 2T units of time

Therefore,

$$\bar{x}_1 = \bar{t} + p_2[(2T + \overline{\max_{N-1}}[\bar{t}])p_3 + 2T(1 - p_3)] =$$

$$= \bar{t} + p_2[2T + p_3\overline{\max_{N-1}}[\bar{t}]] \qquad \text{(B.20)}$$

Let us now consider the case of a PRECOMMIT which arrives at site i from site j. If all PRECOMMIT messages sent by site j to the remaining N-2 sites are accepted (with probability $p_3^{\frac{N-2}{N-1}}$), then, the resources will be locked at site i during $2T + \overline{\max_{N-1}}[\bar{t}]$ units of time. If, on the other hand, any of the remaining N-2 sites rejects the PRECOMMIT message, the resources at site i will be held locked during T (the time to receive the REJECT message). Hence

$$\bar{x}_2 = (2T + \overline{\max_{N-1}}[\bar{t}])p_3^{\frac{N-2}{N-1}} + T(1 - p_3^{\frac{N-2}{N-1}})$$

$$= T + [T + \max_{N-1}[\bar{t}]]p_3^{\frac{N-2}{N-1}} \qquad \text{(B.21)}$$

We also have that

$$\lambda = \lambda_1 + \lambda_2 = \frac{\lambda_u}{p_2 p_3} + (N - 1)\lambda_u p_3^{\frac{2-N}{N-1}} \qquad \text{(B.22)}$$

and

$$\bar{x} = \frac{\lambda_1}{\lambda}\bar{x}_1 + \frac{\lambda_2}{\lambda}\bar{x}_2 \qquad \text{(B.23)}$$

Finally,

$$\lambda\bar{x} = \frac{\lambda_u}{p_2 p_3}\{\bar{t} + p_2[2T + p_3\overline{\max_{N-1}}[\bar{t}]]\} +$$

$$+ (N - 1)\lambda_u p_3^{\frac{2-N}{N-1}}\{T + [T + \overline{\max_{N-1}}[\bar{t}]]\,p_3^{\frac{N-2}{N-1}}\} \qquad \text{(B24)}$$

<u>Proof of Result 6</u>: Let T_i be an update transaction originated at site i. In order for another update transaction T_j to force transaction T_i to be rejected, the following conditions must be satisfied:

1. T_j must arrive at a node j ($j \neq i$). If T_j arrived at site i it would have found the resources locked by T_i .
2. T_j and T_i must conflict.
3. the timestamp of T_j must be less than that of T_i.
4. the PRECOMMIT message of T_j must arrive at site i during the read phase of transaction T_i .
5. the PRECOMMIT of T_j must be accepted at site i.

The average arrival rate γ of T_j type transactions at site j is given by

$$\gamma = \frac{\lambda_u}{p_3} PC \, p_3^{\frac{1}{N-1}} = \lambda_u PC \, p_3^{\frac{2-N}{N-1}} \qquad \text{(B.25)}$$

Let $\tilde{x}_i$ and $\tilde{x}_j$ be the r.v.s which represent the time to execute the read and computation phase of transactions T_i and T_j res - pectively. Let δ be the time interval during which T_j is allowed to arrive at site j in order to satisfy conditions 3 and 4 above. From figure B.1 we have that

$$\delta = \tilde{x}_j + \tilde{x}_i - \tilde{x}_j = \tilde{x}_i \qquad \text{(B.26)}$$

We make here the assumption that $\tilde{x}_j$ and $\tilde{x}_i$ are exponentially distributed with mean $\bar{t}$.

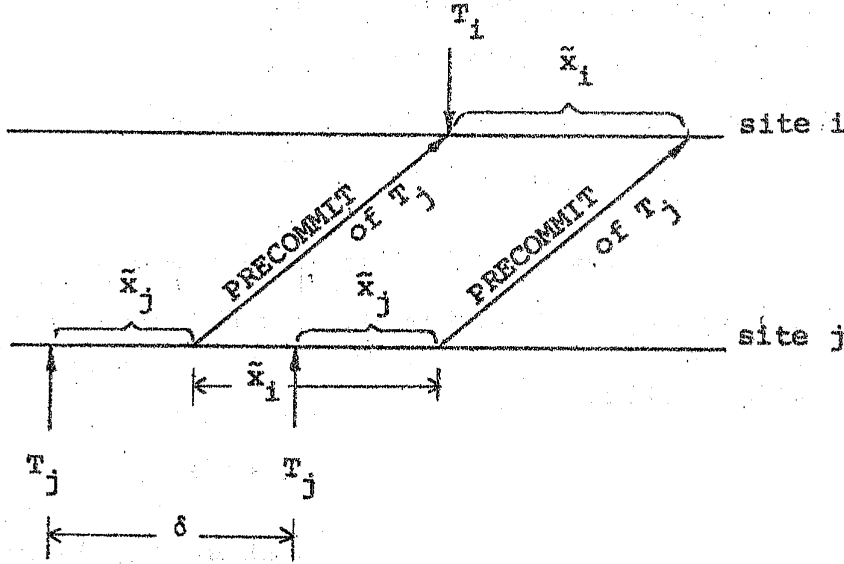


Figure B.1

Therefore, the probability that transaction T_i is not rejected before it enters its precommit fase due to T_j type transactions at site j, given that $\tilde{x}_i = x$ is given by

$$p_2^{\frac{1}{N-1}}|\tilde{x}_1 = x = e^{-\gamma x} \tag{B.27}$$

Unconditioning on $\tilde{x}_1$ we get

$$p_2^{\frac{1}{N-1}} = \int_0^\infty e^{-\gamma x} \frac{1}{\bar{t}} e^{-x/\bar{t}} dx =$$

$$= \frac{1/\bar{t}}{\gamma + 1/\bar{t}} \tag{B.28}$$

Finally, we get

$$p_2 = [\frac{1}{1 + \gamma\bar{t}}]^{N-1} \tag{B.29}$$

Proof of Result 7: An update transaction T_i, originated at site i, can only be rejected during its precommit phase by another update transaction T_j which satisfies the following conditions:

1. T_j must be originated at site j ($j \neq i$)
2. T_j must conflict with T_i
3. the timestamp of T_j must be less than that of T_i
4. the PRECOMMIT message of T_j must arrive at site i after T_i sent its PRECOMMIT messages.
5. the PRECOMMIT of T_j must not be rejected at site i.

Let $\tilde{x}_i$ and $\tilde{x}_j$ be as in the proof of the previous result. In order for conditions 3 and 4 to be satisfied, T_j transactions must arrive at site j during the time interval of duration δ indicated in figure B.2.

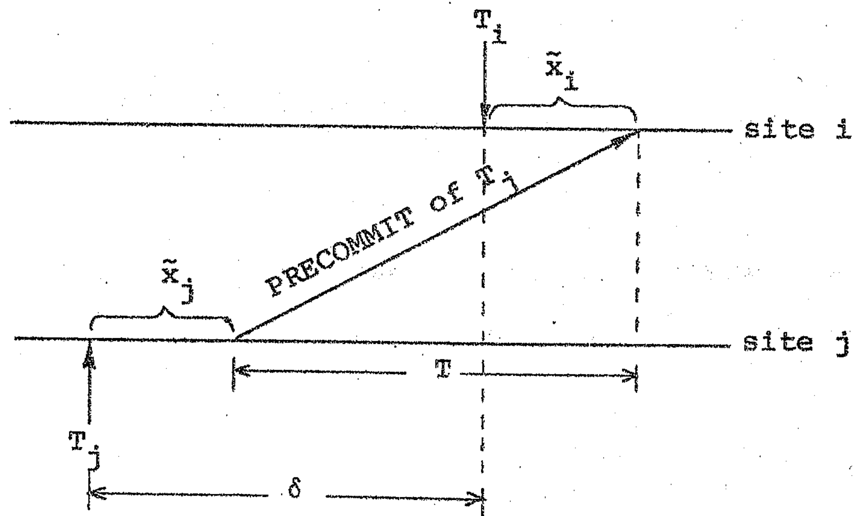


Figure B.2

$$\delta = \tilde{x}_j + T - \tilde{x}_i \qquad (B.30)$$

The arrival rate, α, of T_j type transactions at site j is given by

$$\alpha = \frac{\lambda_u}{p_3} PC \, p_3^{\frac{1}{N-1}} = \lambda_u PC \, p_3^{\frac{2-N}{N-1}} \qquad (B.31)$$

The probability that T_i is not rejected during its precommit phase due to T_j type transactions entering at site j, given that $\tilde{x}_i = x$ and $\tilde{x}_j = y$ is

$$p_3^{\frac{1}{N-1}} | \tilde{x}_i = x, \tilde{x}_j = y = e^{-\alpha(T + y - x)} \qquad (B.32)$$

Assuming, as in the previous result, that $\tilde{x}_i$ and $\tilde{x}_j$ are exponentially distributed with mean $\bar{t}$, we can uncondition on $\tilde{x}_i$ and $\tilde{x}_j$

$$p_3^{\frac{1}{N-1}} = \int_{x=0}^{\infty} \int_{y=0}^{\infty} e^{-\alpha(T+y-x)} \frac{1}{\bar{t}} e^{-x/\bar{t}} \frac{1}{\bar{t}} e^{-y/\bar{t}} dxdy = \qquad (B.33)$$

$$= \frac{1}{(\bar{t})^2} e^{-\alpha T} \cdot \frac{1}{\alpha - 1/\bar{t}} [e^{x(\alpha - 1/\bar{t})}]_0^{\infty} \cdot \frac{(-1)}{\alpha + 1/\bar{t}} [e^{-y(\alpha + 1/\bar{t})}]_0^{\infty}$$

Since we are considering that the system is in equilibrium we must have $\alpha < 1/\bar{t}$. Therefore,

$$p_3^{\frac{1}{N-1}} = \frac{e^{-\alpha T}}{1 - (\alpha \bar{t})^2} \qquad (B.34)$$

Finally,

$$p_3 = \left[\frac{e^{-\alpha T}}{1 - (\alpha \bar{t})^2}\right]^{N-1} \qquad (B.35)$$

ACKNOWLEDGEMENT

The authors would like to thank Marco Antonio Casanova for the helpful discussions they had and for his comments on a draft of this paper.

REFERENCES

[1] Ellis, C.A., "A Robust Algorithm for Updating Duplicated Databases", Proceedings 1977, Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory , University of California, Berkeley California , May 1977.

[2] Thomas, R.H. "A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 5, 2, June 1979.

[3] Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems, 3, 2, June 1978.

[4] Alsberg, P.A., Belford, G.G., Day, J.D. and Grapa, E., "Multi-Copy Resiliency Techniques", Center for Advanced Computation, AC Document Nº 202, University of Illinois at Urbana-Champaign, Urbana Illinois, May 1976.

[5] Menascé, D.A., Popek, G.J. and Muntz, R.R., "A Locking Protocol for Resource Coordination in Distributed Databases", ACM TODS, Jun 1980.

[6] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, IEEE Transactions on Software Engineering, SE-5, 3, May 1979.

[7] Bernstein, P.A., Shipman, D.W. and Rothnie Jr, J.B., "Concurrency Control in a System for Distributed Database (SDD-1)", ACM Transactions on Databases Systems 5, 1, Mar 1980.

[8] Reed, D.P., "Naming and Synchronization a Decentralized Computer System", Ph.D. Thesis, M.I.T. Department of Electrical Engineering, Sep 1978.

[9] Bernstein, P.A., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", Proc. 6th VLDB Conference, Montreal, Canada, Oct 1980.

[10] Bernstein, P.A., and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Tech. Rep., Computer Corporation of America, Feb 1980.

[11] Garcia Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database" Ph.D. Dissertation, Computer Science Department, Stanford University, Jun 1979.

[12] Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System", Proc. 4th Berkeley Conference on Distributed Data Management & Computer Networks, Aug 1979.

[13] Dantas, J.E.R.,"Performance Analysis of Distributed Database Systems "Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, 1980.

[14] Menascê, A.D. and Nakanishi, T., "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", Information Systems, 7, 1.

[15] Hevner, R. and Bing Yao, S., "Query Processing in Distributed Database Systems", IEEE Trans. on Software Engineering SE-5, 3, May 1979.

[16] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Data Storage System", Tech. Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California 1976.

[17] Menascé, D.A., and Landes, O., "On the Design of a Reliable Storage Component for Distributed Database Systems", ibid [9] .

[18] Gardarin, G. and Chu W.W., "A Distributed Control Algorithm for Reliably and Consistently Updating Replicated Databases", IEEE Transaction on Computers, C-29, 12, Dec. 1980.

[19] Lee, H., "Queueing Analysis of Global Locking Synchronization Schemes for Multicopy Databases", IEEE Transaction on Computers, C-29, 5, May 1980.

[20] Kleinrock, L., "Queueing Systems", Vol. 2, John Wiley & Sons, 1976.

[21] Jackson, J.R., "Networks of Waiting Lines", Operations Research, 5, 1957.

[22] Kleinrock, L., "Queueing Systems", Vol. 1, John Wiley & Sons, 1975.