205

PERFORMANCE EVALUATION OF CONCURRENCY CONTROL ALGORITHMS IN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

BY

THOMAS MICHAEL SIEBEL

A.B., University of Illinois, 1975
M.B.A., University of Illinois, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

Q.004 Ta5Si   Engineering

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1985

WE HEREBY RECOMMEND THAT THE THESIS BY

THOMAS MICHAEL SIEBEL

ENTITLED PERFORMANCE EVALUATION OF CONCURRENCY CONTROL ALGORITHMS IN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Geneva Belford
Director of Thesis Research

Charles W Gear
Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

O-517

University of Illinois at Urbana-Champaign

DEPARTMENTAL FORMAT APPROVAL

THIS IS TO CERTIFY THAT THE CONTENT, FORMAT, AND QUALITY OF PRESENTATION OF THE THESIS SUBMITTED BY THOMAS MICHAEL SIEBEL AS ONE OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IS ACCEPTABLE TO THE DEPARTMENT OF COMPUTER SCIENCE

Department/Division/Unit

April 29, 1985
Date of Approval

Dorothy A. McLaughlin
Departmental Representative

TABLE OF CONTENTS

CHAPTER 1
DISTRIBUTED DATABASES

Distributed database management systems provide a solution to some very real problems for a geographically distributed organization requiring a unified information sharing and processing system.

Traditional database management system (DBMS) technology is centralized. In a centralized database all of the data is stored and processed at a common location. Remote access to the DBMS is accomplished via geographically dispersed terminals.

In a distributed database the data is partitioned across a number of (possibly geographically dispersed) locations. The database may be fully replicated at each node, fully partitioned across the nodes into disjoint subsets, or partially replicated at each node. A processor with local storage is located at each node. A communications network is provided to link the nodes. Generally, any node may access data residing at any other node. [Bray, 76]

A major objective of a distributed DBMS is to provide location and replication transparency. Location transparency means that users should not need to know at which site or sites any particular data items are stored. Data replication means that any logical data item can have several stored representatives at various sites. Ideally, all details of data location and maintenance should be handled by the system, not the user. Location and replication transparency together imply that the distributed DBMS should look like a centralized DBMS to the user. [Date, 83]

1.1 Advantages of Distributed Database Systems

Distributed databases potentially offer increased performance over traditional technologies while minimizing system costs. A distributed database management system tends to be more reliable than a centralized DBMS because it is composed of multiple computers at multiple locations. The distributed DBMS is subject to gradual performance degradation as nodes or links fail.

An additional level of fault tolerance is afforded when the database is distributed in such a fashion that individual data items reside at more than one node simultaneously. Data redundancy in the system provides an alternative access path for any data item should any node fail.

Because the data and processing power in a distributed DBMS is available locally, response time can be minimized and telecommunication costs reduced. In recent years data processing and storage costs have been decreasing at roughly twice the rate of decreases in data communications costs. This trend is making it increasingly attractive to substitute lower local processing and storage costs for relatively costly data communications facilities. [Champine, 77]

Due to the modular structure of a distributed DBMS, system enhancement and expansion costs can be minimized. Incremental upward scaling of the system is facilitated through modular reconfiguration, modular implementation and modular upgrade. [Champine, 77]

Composed of multiple processors, a distributed DBMS can offer greater processing, access and data capacity than any single machine.

Distributing a system provides an increased level of autonomy to the various subdivisions within an organization. Individual groups can exercise local control over their own data, with local accountability, and less dependence on a remote data processing center. [Date, 83]

In sum, a distributed DBMS offers improved performance, improved system availability, fault tolerance, gradual degredation, easier adjustment to a growing or changing workload, local autonomy, greater capacity, and operating cost economy.

1.2 Problem Areas in Distributed Data Base Management Systems

In a centralized DBMS the primary performance objective is the minimization of secondary storage access time. In a distributed DBMS however, the primary objective is the minimization of the volume of data and number of messages transmitted across the communications network. [Bray, 76]

In general, the bandwidth of a communication link is much less than that of secondary storage devices. While a communication link (e.g. Arpanet) will typically transmit data at roughly 25,000 bytes per second, a typical disk drive will operate at a data rate on the order of one million bytes per second. In addition, communication networks typically have slow access times, with delays on the order of 100 milliseconds. [Date, 83]

Aside from access and propagation delays, considerable CPU overhead is introduced by the internode message handling requirements of a distributed DBMS. To send a message between nodes and handle the acknowledgement will typically require between 5,000 and 10,000 instructions of operating system and communications control code. [Date, 83]

Node and link failures in a distributed DBMS introduce the potential for the database system to degenerate into a partitioned state--the total network split into a collection of disjoint sub-networks. Nodes may remain operational, yet unable to communicate with other nodes.

Additional problems arising from distributed DBMS's include:

* Concurrency Control

* Recovery

* Query Processing Optimization

* Deadlock

* Commit Protocols

* Catalog Management

This paper will specifically address the issue of concurrency control in distributed DBMS. Although related problems such as deadlock and commit protocols will be touched upon as necessary, they will not be discussed in depth.

1.3 Concurrency Control

Concurrency control is concerned with the problem of coordinating concurrent accesses to a database in a multiprogramming system. Concurrency control permits an increased degree of multiprogamming while providing the user with a seemingly dedicated system.

Concurrency control algorithms maintain database integrity by preventing database updates performed by one user or process from interfering with the data retrievals and updates of another. [Bernstein and Goodman, 81]

Concurrency control in centralized DBM's has been an active research area for a number of years and is well understood. Distributed DBMS concurrency control is a more recent problem and is in a state of flux.

The concurrency control problem is exacerbated in the distributed case as 1) multiple users can simultaneously access the data at (and from) multiple locations and 2) propagation delays introduced by the distributed system preclude the concurrency control mechanism at one node from instantaneously knowing about information at other nodes of the system. [Bernstein and Goodman, 81]

The goal of concurrency control is to prevent interuser interference during the simultaneous update or retrieval of data. Interuser interference may take the form of 1) Lost Updates or 2) Inconsistent Retrievals.

1.4 The Lost Update Anomaly

This form of interference may occur when two transactions are attempting to simultaneously update the same data. Two such transactions could read the data value roughly simultaneously, compute the new values in parallel, and then write the new values to the database. The net effect is incorrect -- the database reflecting the activity of only one transaction.

Lost Update Anomaly

Transaction 1	Database	Transaction 2
	X = 100	
read(X)		read(X)
compute X = X + 50		compute X = X + 100
write(X)	X = 150	
	X = 200	write(X)

In this example, the effect of transaction 1 has been "lost" or overwritten by transaction 2. The update of transaction 1 is "lost" and database integrity has not been maintained, the database having been left with an erroneous value for X. All subsequent transactions manipulating or dependent upon data item X will produce incorrect results, further eroding database integrity.

1.5 Inconsistent Retrieval Anomaly

In this example we consider a situation where two interleaved transactions are accessing the same data. If transaction 1 (although correct in itself) leaves the database temporarily inconsistent, transaction 2 may read the inconsistent data and produce erroneous results.

Inconsistent Retrieval Anomaly

Transaction 1	Database	Transaction 2
	X = 100	
	Y = 200	
read(X)		
compute X = X - 50		
write(X)	X = 50	
	Y = 200	read(X)
read(Y)		read(Y)
compute Y = Y + 50		compute Z = X + Y
write (Y)	X = 50	write (Z)
	Y = 250	
	Z = 250	

In this example, transaction 1 decrements and increments two values by equal amounts. Transaction 2 computes the sum of these two values. As shown, the interleaving of these transactions results in an incorrect value being calculated for variable Z--the sum of values X and Y. The calculated value of Z is 250, the actual value of Z should be 300.

CHAPTER 2

TRANSACTION PROCESSING MODEL

Our transaction processing model is based on the Bernstein and Goodman model of concurrency control in a distributed DBMS. [Bernstein and Goodman, 81]

In this model processes interact with the distributed DBMS by executing transactions. Given an initially correct database state, an individually correct transaction will maintain database integrity if executed in isolation. We assume throughout this paper that each transaction represents an individually complete and correct computation.

Our model provides for four types of components: transactions, transaction managers (TM's), data managers (DM's), and data.

Transactions are issued by the various processes at the nodes. Associated with each node is a TM, a DM, and data. Transactions communicate with TM's, TM's communicate with DM's, and DM's manage the data.

TM's supervise the transactions. Each transaction is supervised by the TM at the site of transaction generation. Four operations are defined in the transaction-TM interface: Begin, Read(X), Write(X,newvalue), and End. [Bernstein and Goodman, 81]

The function of each component is described below:

Begin:

The TM initializes a private workspace for the transaction.

Read(X):

The TM examines the local workspace for a copy of X. If the data is not available in the workspace, the TM selects some stored value of X (X_i) in the database and issues DM-read(X_i) request to the appropriate DM. When the data becomes available, the TM places a copy of X in the transaction's workspace, returning the value of X to the transaction.

Write(X,newvalue):

The value of X in the transaction's private workspace is updated to reflect the new value.

End:

A two-phase commit protocol is initiated to update the database. For each X updated by the transaction and for each copy X_i of X in the distributed database, the TM issues a DM-prewrite(X_i) to the DM that manages X_i. After all DM-prewrites for a value X are acknowledged, the TM issues DM-writes for each X_i. The transaction then terminates.

2.1 Serializability

Our transaction model assumes that any transaction executed in isolation will maintain database integrity. It follows that any serial execution of a given set of transactions (i.e. any execution of the transactions one at a time, in any order) will also preserve database integrity. Any serial execution is therefore correct.

An execution sequence is said to be serializable if it is computationally equivalent to a serial execution--that is if it produces the same results as some serial execution sequence operating upon the same initial database.

Serializability is the formal criterion for correctness in transaction execution. A given interleaved sequence of transactions will be considered correct if and only if it is serializable. The goal of concurrency control is to ensure that all transaction execution sequences are serializable. [Date, 83]

DM-read's and DM-write's are the only operations that actually result in database access. Thus to guarantee serializability, it is sufficient to control the execution sequence of transactions by the execution of DM-read and DM-write operations at the various DM's of the system.

2.2 Implementation of Concurrency Control

Numerous concurrency control algorithms have been reported in the literature. Most of them employ combinations (or variants) of two basic algorithms: two-phase locking (2PL) and basic timestamp ordering (BTO). Both 2PL and BTO guarantee serializability. [Bernstein and Goodman, 81]

2.3 Two-Phase Locking

Two-phase locking (2PL) serializes transactions by detecting conflicts between read and write operations and preventing their concurrent execution. Two operations are said to conflict if both attempt to access the same object and one of them is a write operation.

If one transaction issues a DM-read(X) and another issues a DM-write(X), the value read by the DM-read(X) will, in general, depend upon the order in which the operations were executed. This is called read-write(rw) conflict. This type of conflict is exemplified by the inconsistent retrieval anomaly described above.

If two transactions issue simultaneous DM-write(X) requests, the ultimate value of X in the database will be dependent upon the ordering of execution of the operations. This is called write-write(ww) conflict.

As its name implies, 2PL employs a locking scheme to detect and prevent conflicts. Before reading an object X, a transaction must be granted a read-lock on that object. Before writing to X, it must own a write lock on X.

Lock ownership is governed by two rules:

1) Different transactions cannot simultaneously own conflicting locks.
2) Once a transaction releases a lock, it cannot obtain additional locks.

If all transactions follow rules one and two then all interleaved executions of those transactions are serializable. Rule 1 is sufficient to avoid both ww and rw conflict. Rule 2 causes transactions to obtain locks in a two-phase manner. During the "growing phase" a transaction acquires all its locks. The "shrinking phase" begins when a transaction releases its first lock. During the shrinking phase the transaction will release all of its locks and is prevented from acquiring additional locks.

2PL is implemented by a software module called a 2PL scheduler. In basic 2PL the scheduler is distributed along with the data. Transactions request locks by issuing DM-read's and DM-write's. If the requested lock cannot be granted, the scheduler places the request on a queue associated with that data object until the conflict is resolved.

As in any similar locking scheme, deadlock can result. Various approaches have been developed to deal with the deadlock problem. Either deadlock prevention or deadlock detection can be employed. [Chu and Ohlmacher, 74]

In basic 2PL a transaction need only obtain a readlock on any one copy (X_i) of (X) in the database. To execute a DM-write however, the transaction must obtain writelocks on all copies of X.

2PL will guarantee that the concurrent execution of transactions is equivalent to some (unpredictable) serial execution of the transactions. [Date, 83]

One of the drawbacks of 2PL is the amount of message traffic required. An update of an object that exists at N locations will require 5N messages. [Date, 83]

N lock requests

N lock grants

N update messages

N acknowledgements

<u>N lock releases</u>

5N Total Messages

Three variants of basic 2PL have been proposed with the intent of reducing message overhead. These variants include: Primary Copy 2PL, Voting 2PL, and Centralized 2PL.

2.4 Primary Copy 2PL

In this implementation, one copy (X_i) of each object (X) in the database is designated the primary copy. Before accessing any object, a transaction must obtain an appropriate lock on the primary copy.

Primary copy 2PL increases the message overhead for a readlock, but significantly reduces the overhead for a writelock. For a readlock, if a transaction wishes to read a copy of an object other than the primary copy, it must now communicate with two DM's -- the DM where the object is stored and the DM where the primary copy resides. In the writelock case however, the number of lock requests, acknowledgements and releases is reduced by 3(N-1). Under Primary Copy 2PL, 2N+3 messages are required to update an object.

1 lock request

1 lock grant

1 lock release

N update messages

<u>N acknowledgements</u>

2N+3 Total Messages

2.5 Voting 2PL

Voting 2PL is a variant of the two-phase locking algorithm that exploits data redundancy to expedite transaction execution. Voting 2PL is suitable for ww synchronization only.

Using Voting 2PL the TM requests a writelock from each DM holding a copy of X. Each DM responds immediately with a "lockset" or "lockblocked" acknowledgement. Upon receipt of the acknowledgement from the DM's, the TM counts the number of "lockset" messages. If the number of "lockset" acknowledgements constitutes a majority of the messages received, the TM behaves as if all locks were set and completes execution of the transaction.

If a majority is not attained, the TM awaits receipt of a sufficient number of "lockset" messages to constitute a majority.

As only one TM can obtain a majority of the "lockset" messages at any given time, only one transaction can be in the second phase of the commit cycle simultaneously.

The effect of Voting 2PL is to minimize queueing delays during lock negotiation.

2.6 Centralized 2PL

This implementation employs a centralized scheduler to handle all lock requests and acknowledgements. Like Primary Copy 2PL, this scheme will increase communications requirements for readlocks but reduce message overhead for writelocks. Again, 2N+3 messages are required.

Recent simulation studies suggest that substantial performance gains are afforded by centralized 2PL over basic 2PL. [Garcia-Molina, 79]

The drawbacks of Centralized 2PL are that the central locking site is likely to become a system bottleneck and that system reliability is reduced. If the central locking site fails, the entire system will fail.

2.7 Basic Timestamp Ordering

Using BTO, each transaction is identified by a unique global timestamp. The DM's are required to process conflicting operations in timestamp order.

The essential difference between timestamping and locking techniques is that while locking results in a transaction interleaving equivalent to some serial execution of the transactions; timestamping results in an execution sequence equivalent to a specific serial execution of transactions--namely, that execution sequence defined by the chronological order of the timestamps.

Using BTO, conflicts are resolved by restarting the conflicting transaction. Restarted transactions are issued a new timestamp. The execution sequence resulting from BTO is that of the chronological ordering of successfully completed transactions.

Unique timestamps are assigned in a synchronous distributed system by the concatenation of the local clock time at transaction generation time and the local node id. Associated with each data object are two timestamp values TS(r) and TS(w). TS(r) is set to the timestamp of the transaction that last read the object. TS(w) is set to the timestamp of the transaction that last updated the object.

A conflict situation occurs when a transaction (T_i) attempts to read or update an object with a timestamp greater than its own. The two conflict situations occur when $TS(T_i) < TS(r)$ or $TS(T_i) < TS(w)$. If a conflict occurs, the conflicting transaction is restarted and assigned a new timestamp.

Timestamping presents some advantages over locking. As no locks are necessary, deadlock is not a possibility and the communication overhead of locking and deadlock prevention or detection is avoided. Disadvantages arise from the overhead and transaction delays resulting from conflict detection and transaction restart.

2.8 Thomas Write Rule (TWR)

The Thomas Write Rule is a ww synchronization technique that precludes transaction restart. Using this technique update conflicts are resolved by simply ignoring the update request of the conflicting transaction. The transaction is not restarted. This technique assumes that the update request of a conflicting transaction where ($TS(T_i) < TS(w)$) contains obsolete information and thus can be ignored. TWR guarantees that the effect of DM-writes upon the database is equivalent to that of a sequence of DM-writes in timestamp order.

2.9 Conservative Timestamp Ordering (CTO)

Conservative Timestamp Ordering eliminates the need for transaction restart in both ww and wr conflicts. The fundamental idea in CTO is that no read or update operation is performed until it can be guaranteed that no subsequent conflict will occur.

CTO requires that DM-reads and DM-writes be performed in timestamp order. If an operation arrives that might cause future conflicts (i.e. arrives out of timestamp order) the system delays that operation until it is sure that no conflicts are possible.

Using CTO, it is no longer necessary to associate a TS(w) and TS(r) with each data object, thus storage requirements are reduced. Communication overhead is increased however, as it is necessary for each scheduler to be in constant communication with all other schedulers to ensure sequential execution. Another disadvantage of CTO is that it reduces the degree of system concurrency. CTO eliminates the possibility of conflict by serializing all operations at each site, not just those operations that conflict.

Our simulation model attempts to replicate the operation of a distributed concurrency control algorithm in terms of the events of the individual elements of the system. The interrelationships of the elements are built into the model. The simulation model allows the computer to capture the effects of the elements' actions on each other as a dynamic process.

We feel that simulation is particularly well suited to the evaluation of concurrency control algorithms. Concurrency control in a simulation model can be implemented with the identical software modules that might be employed in an actual DBMS. Simulation provides the ability to monitor algorithm performance while controlling external factors such as transaction volume, data distribution, and deadlock detection overhead.

3.2 Simulation Environment

Our experiment was conducted at the University of Illinois Computing Research Laboratory in Urbana, Illinois. The purpose of our study was to design and implement a computer simulation model of a distributed database management system. To enforce concurrency control, two schedulers were implemented and evaluated. The two software schedulers implemented the 2PL and BTO algorithms.

A number of simulation languages were evaluated for the development of our model including Path Pascal, Simula and GPSS. Path Pascal was selected for use as it seemed to most readily lend itself to the simulation of concurrent processes. [Kolstad and Campbell, 80]

The simulators were written, debugged, verified in the period of September 1983 to May 1984. The simulations were run on a dedicated Vax 11/750 computer under the Berkley Unix 4.1 operating system.

3.3 System Architecture

We attempted to make our simulator as independent as possible of network topology, message handling delays and other implementation specific parameters. The rationale here was to make the simulation results a function of only the concurrency control algorithms themselves--not secondary factors. Our simulation has no message queues or communication delays other than those specific to the individual algorithms.

Our model presumes a reliable communication network. We are not concerned with the specific network topology. We avoid dealing with the variable propagation delays introduced by network overhead and by the distances between nodes. In our model, communications are assumed fully reliable with messages arriving virtually instantaneously at their destinations. As node failure is precluded and all messages are guaranteed to arrive, we avoid issues of database and transaction recovery.

3.4 Database Design

Our database consists of three nodes, each of which contains one hundred data objects. The three node case is of sufficient complexity to present all of the problem and conflict situations that arise in a distributed DBMS, and at the same time present a manageable problem in terms of machine requirements to run the simulations. We chose to model a fully partitioned distributed database, with no data object stored at more than one node. Any node may issue transactions to any other node. Each transaction references objects specific to a single node.

For our purposes the "data object" may be considered as the unit of information being locked. Although we do not deal with the issue of locking granularity directly in our simulation, the level of locking could take place at the table, page, data structure, record, or data item. A relational implementation of the database system also provides the potential for predicate locking. As the level of locking granularity decreases, i.e. as we lock smaller units of information, the concurrency control problem becomes more complex and the degree of concurrent access to the database increases.

Subsequent to transaction generation, all transaction processing occurs at the node where the data is stored. This simplification avoids the complexity and overhead of additional algorithms to deal with cataloging and directory distribution. By controlling this factor we should be able to evaluate the concurrency control algorithms in relative isolation, while generating a realistic frequency of transaction conflict conditions.

We assume that the data base is distributed in such a manner that there is an 80% probability that a transaction will reference data local to its node of origin. We feel this is a realistic assumption as one of the primary advantages and objectives of a distributed DBMS is the minimization of communication costs and propogation delays. Therefore, any actual data distribution technique would attempt to maximize the local accessibility of data.

3.5 Transaction Processing Modelling

Our transaction processing model is patterned after the Bernstein and Goodman model described in chapter 2. Each node contains a transaction generator, a transaction manager, and a data manager. There is no actual database in our model being read, processed or updated. The delays introduced by actual read, write, and processing operations are not accounted for -- being independent of the concurrency algorithm and therefore controllable. Our concern is exclusively with the delays and overhead introduced by the necessity for concurrency control.

The data manager at each node maintains a data structure appropriate to control access to the database objects, dependent upon the algorithm employed.

For BTO an array is employed to record the read and write timestamps for each data object. The timestamp of the requesting transaction is compared to the appropriate timestamp of the data object.

If the transaction timestamp is greater than the appropriate read or write timestamp for all the objects requested, the object timestamps are set to that of the transaction and the transaction executes to completion. If not, the transaction is restarted.

To implement 2PL, two fifo queues are maintained for each data object in the form of a linked list. One queue is used to store write lock requests, the other stores read requests. A counter is maintained for each queue to store its length. To negotiate each lock request, the appropriate counters are examined to determine whether or not the lock can be granted. If it cannot be granted the request is added to the back of the queue. As locks are released, the counters are again examined to determine whether or not another enqueued lock request can be granted.

3.6 Transaction Generation

The transaction generator located at each node of the system simulates the actual processes at each node that would be issuing transactions.

We assume that interarrival times of transactions are exponentially distributed. Given an "arrival," we use a uniform random variable to determine whether or not the read and write requests are local. We assume that 80% of the transaction requests are local to the node of transaction origin. If a transaction is not local, a uniform random variable is generated to determine the node to be accessed, with each of the two other nodes having an equal probability of selection.

The number of data objects accessed by each transaction is again a randomly distributed variable. We model this as a "shifted" poisson process with a mean of 4. We eliminate the possibility of a transaction attempting to access zero objects by "shifting" the distribution--i.e. incrementing the result by one. Thus the number of data objects accessed by any transaction is poisson(3) + 1. We assume that 30% of the object requests are for write operations.

Our model contains 100 data objects located at each node of the system. We assume that the data objects are not in uniform demand. In selecting the individual objects to be accessed, we assume that 20% of the objects receive 80% of the access requests. We feel a uniform distribution of lock request across the database would be unrealistic and would tend to minimize transaction conflicts. In this manner, after determining the number of read and write requests of a transaction and the node to be accessed, we generate a geometrically distributed random variable with mean = 0.0957 as the first object to be accessed. All subsequent objects referenced by that transaction are sequential. In an effort to structure the effect of data locality into our model, we have each transaction access sequential data addresses with 30% of the accesses, on the average, write operations.

Upon completion of transaction generation, the transaction is forwarded to the appropriate transaction manager for negotiation.

3.7 Experimental Design

A synchronous clock is maintained in our simulator. One unit of time is associated with the execution of one program instruction. Processing delays for each algorithm are simulated by incrementing a counter one unit for each line of code processed in the negotiation of that transaction. The clock is incremented only for that code encountered in the concurrency control portion of the code. To record response time, the clock "starts" immediately after transaction generation and stops after the transaction has completed successfully.

Fifty transactions were generated at each of the three nodes of the system and run to completion. The simulation was run four times for each algorithm. The mean interarrival time at transaction generation was varied to simulate various levels at system load. Statistics were gathered for the average response time at each run as measured by the number of lines of code executed in the negotiation of the 150 transactions.

A lower bound of 90 was determined through experimentation to be reasonable lower bound for interarrival times. By running extremely light loads (i.e. lengthy interarrival times) through both simulators we found that the 2PL simulator exhibited an average response time of about 90, this was significantly greater that that for BTO.

It became apparent that to employ an interarrival time less than 90 in the 2PL simulator would, in theory, result in infinite queueing delays--essentially analogous to a queueing system in which the interarrival times are less than the mean service time.

After achieving stability in simulator behavior, both simulators were run four times with mean interarrival times varying from 90 to 150. This sequence was repeated using three different random number seeds for the random number generator driving each transaction generator.

AVERAGE TRANSACTION RESPONSE TIME

	Average Transaction Interarrival Time	150	125	100	90
Algorithm:	2PL	92.3	93.6	98.6	105
	BTO	57.6	59.7	62.4	63.6

3.8 Conclusion

The results of our simulation indicate that, at all levels of system load the basic timestamp ordering algorithm exhibits superior performance as measured by average transaction response time.

These results seem attributable to the transaction restart overhead of BTO being less than that required for locking and queue maintenance of 2PL.

A comparison of the storage requirements and data structure complexity of BTO and 2PL suggests that BTO affords some significant advantages. The substantially greater search time and maintenance overhead of 2PL suggest that BTO should exhibit superior performance.

For BTO, the data structures required consist of one read timestamp and one write timestamp for each database object, totalling 2N data structures, where N is the number of objects in the database.

The storage overhead for 2PL is much greater than that of BTO. At a minimum, with no lock requests outstanding, the 2PL requires 6N storage structures for lock maintenance overhead.

N	- Outstanding readlock counters
N	- Outstanding writelock counters
N	- Head of queue pointers for writelock queues
N	- Tail of queue pointers for writelock queues
N	- Head of queue pointers for readlock queues
$\underline{N}$	- Tail of queue pointers for readlock queues
6N	- Minimum structure size

In addition, the size of the storage structures for 2PL (and magnitude of associated maintenance and searching overhead) increases dramatically as transaction conflicts are incurred and requests become enqueued.

Total Storage Required $$6N + \sum_{i=1}^{N}(W_i) + \sum_{i=1}^{N}(R_i)$$

N = Number of data objects in database

W_i = Number of enqueued writelock requests for the i^{th} object

R_i = Number of enqueued readlock requests for the i^{th} object

We can see that the storage and maintenance overhead of 2PL will increase dramatically as the number of objects (N) and transaction volume (outstanding lock requests) increases. The relative performance disparity of the BTO and 2PL may well become more pronounced as these parameters grow.

APPENDIX A

TWO-PHASE LOCKING SIMULATOR

```
(*************************************************************************)
(*                      BASIC TWO-PHASE LOCKING                          *)
(*************************************************************************)

program concurrency (input,output,ddbug);


   const   (*************************************************************)
      tranmax = 40;         (* max number of variables accessed *)
      n = 100;              (* number of dataobjects at each node   *)
      landa2 = 5;
      wpct = 0.3;           (* percent of accesses that are writes *)
      p = 0.0957;           (* mean of geometric distribution   *)
      nodes =2 ;            (* number of nodes(-1) in system   *)
      sseed = 12345;        (* initial seed for random num generator *)
      timelimit = 1;        (* timout for deadlock *)
      maxtid = 1002;          (* max index for arrays *)



   type    (*************************************************************)

      topptr = ^toper;
      toper = record            (* transaction operation *)
         id : integer;          (* data object id *)
         tid : integer;         (* transaction id *)
         tnum : integer;        (* transaction number *)
         typ : char;            (* type of access: write/read *)
         dest : integer;        (* access node id *)
         locked : boolean;      (* lock grant flag for trans  *)
         next : topptr;         (* pointer *)
         end;

      transptr = ^transrec;
      transrec = record                 (* transaction record *)
         nodeid : integer;              (* source node id *)
         destnode : integer;            (* destination node id *)
         restart : boolean;             (* restart flag *)
         tr_ts : integer;               (* transaction timestamp *)
         lockcount : integer;           (* tot # of locks requested*)
         trid : integer;                (* transaction id *)
         tnum : integer;                (* transaction number *)
         tread : integer;               (* # read locks requested *)
         twrite : integer;              (* # write locks *)
         transblk : array[1..tranmax] of topptr;  (*array of operations *)
         end;



      (*object fifoqueue is used for blocked access requests) *)
      fifoqueu = object
         path (enq;deq),1:(putonq,takeoffq,unq) end;

       var
          headofq : topptr;
          tailofq : topptr;

       entry procedure putonq (arrival : topptr);
          begin
             await(time + 4);
             if arrival <> nil
                then arrival^.next := nil;
             if headofq = nil
                then headofq := arrival
             else tailofq^.next := arrival;
         tailofq := arrival;
      end;   (* procedure putonq *)

   entry function takeoffq : topptr;
      begin
         await(time + 2);
         takeoffq := headofq;
         if headofq <> nil
            then headofq := headofq^.next;
      end;   (* function takeoffq *)

   entry procedure enq (arrival : topptr);
      begin
         await(time + 1);
         putonq (arrival);
      end;   (* procedure enq *)

   entry function deq : topptr;
      begin
         await(time + 1);
         deq := takeoffq;
      end;   (* function deq *)

   entry procedure unq (rem : topptr);
      var
         rid : integer;
         last : topptr;
         here : topptr;
         found : boolean;

      begin
         await(time + 4);
         found := false;
         last := headofq;
         if(last <> nil) then
            here := last^.next;
         rid := rem^.tid;

         if(headofq^.tid = rid) then begin
            await(time + 3);
            found := true;
            if(headofq=tailofq) then
               headofq := nil
            else
               headofq := headofq^.next;
            end;
         while (not found) do begin
            await(time + 4);
            if(here^.tid = rid) then begin
               if (here = tailofq) then
                  last^.next := tailofq
               else last^.next := here^.next;
               found := true;
               end;
            if(here = tailofq) then   (* not found condition *)
               found := true;
         end;
      end;

   init;
      begin
         headofq := nil;
         tailofq := nil;
      end;

           end; (* fifoqueu *)
   (*********************************************************************)
var
   ddbug : text;
   ceed : integer;
   runjobs : integer;            (* length of simulation in jobs per node *
   lamdal : integer;
   rscount : integer;            (* # of restarted transactions *)
   stats : object
      path 1:(prt) end;

      var
         tput : real;            (* average throughput *)
         rtime : real;           (* average response time *)
         fincount : real;        (* # completed jobs *)
         trtime : real;          (* response time of indiv transaction *)

      entry procedure prt(xxtion : transptr);
         begin
            fincount := fincount + 1.0;
            tput := fincount/time;
            trtime := time - xxtion^.tr_ts;
            rtime := (((fincount-1)/fincount) * rtime)
                        + ((1/fincount)*trtime);
            writeln(' ---------------------------------');
            writeln('transaction completed at time :  ',time);
            writeln('total jobs completed = ',fincount);
            writeln('tr_nodeid = ',xxtion^.nodeid);
            writeln('tr_timestamp = ',xxtion^.tr_ts);
            writeln('tr_id = ',xxtion^.trid);
            writeln('TR response time = ',trtime);
            writeln('ave response time = ',rtime);
            writeln('ave throughput = ',tput);
            if(trunc(fincount) = (nodes+1) * runjobs) then begin
               writeln('=-== ======two phase locking===========');
               writeln('lamda = ',lamdal);
               writeln('runjobs = ',runjobs);
               writeln('ave response time = ',rtime);
               writeln('ave throughput = ',tput);
               end;
         end;   (* procedure prt *)

      init;
         begin
            tput := 0.0;
            rtime := 0.0;
            fincount := 0.0;
            rscount := 0;
            trtime := 0.0;
         end;
      end; (*object stats *)


   trans_mgr : array[0..nodes] of object
   path     tm  end;

   var

   countlocks : array[1..maxtid] of integer;
   link : array[1..maxtid] of object

      path (sendlock;recvlock) end;
   entry procedure sendlock;

      begin
         (*no-op*)
      end;   (* procedure sendlock *)

   entry  procedure recvlock(tsact:transptr);
      begin
         (* no-op *)
         countlocks[tsact^.trid] := countlocks[tsact^.trid] + 1;
         write(ddbug,'revclock:  countlocks = ');
         write(ddbug,tsact^.trid);
         writeln(ddbug,countlocks[tsact^.trid]);
      end;    (* procedure recvlock *)


   end; (* link object *)

synch : array[1..maxtid] of object

   path (send;recv) end;

   entry procedure send;

      begin
         (*no-op*)
      end;   (* procedure sendlock *)

   entry  procedure recv(tsact:transptr);
      begin
         (* no-op *)
         writeln(ddbug,'-----------SYNCH.RECV------tid = ',tsact^.trid)
      end;    (* procedure recvlock *)


   end; (* synch object *)


 (* dm is the data manager located at each node
 this version of dmgr implements the basic 2pl
 concurrency control algorithm*)

 dm : object

 path 1:(rgrant,wgrant),1:(restart,complete),
        dm_write,dm_read,dm_r_w,getlocks,releaselocks end;

 type
    lockrec = record
       rstat : integer;     (* # of read lock granted *)
       wstat : integer;     (* # of write locks granted *)
       readq : integer;     (* # waiting read req^.ests *)
       writeq : integer;    (* # waiting write req^.ests *)
       r_ts : integer;      (* timestamp of last read *)
       w_ts : integer;      (* timestamp of last write *)
       wque : fifoqueue;    (* waiting write req^.ests *)
       rque : fifoqueue;    (* waiting read req^.ests *)
    end;

  var
    locktbl : array [1..n] of lockrec;  (*one for each data object*)
    commit : array[1..maxtid] of boolean;

   entry procedure rgrant (req : topptr);
    begin
       await(time + 3);
```

```
         write(ddbug,'rgrant:    ');
         write(ddbug,req^.tid);
         write(ddbug,req^.id);
         write(ddbug,' ',req^.dest);
         writeln(ddbug,time);
         locktbl[req^.id].rstat :=
            locktbl[req^.id].rstat + 1;
         req^.locked := true;
         link[req^.tid].sendlock;-
         writeln(ddbug,'rgrant:sendlock: tid = ',req^.tid);
      end;    (* procedure rgrant *)


   entry procedure wgrant (req : topptr);
      begin
         await(time + 3);
         write(ddbug,'wgrant:    ');
         write(ddbug,req^.tid);
         write(ddbug,req^.id);
         write(ddbug,' ',req^.dest);
         writeln(ddbug,time);
         locktbl[req^.id].wstat :=
            locktbl[req^.id].wstat + 1;
         req^.locked := true;
         link[req^.tid].sendlock;
         writeln(ddbug,'wgrant: sendlock--tid = ',req^.tid);
      end;     (* procedure wgrant *)


   entry procedure dm_read (req : topptr);
      begin
         await(time + 2);
         write(ddbug,'dm_read:   ');
         write(ddbug,req^.tid);
         write(ddbug,req^.id);
         write(ddbug,' ',req^.dest);
         writeln(ddbug,time);

         if (locktbl[req^.id].wstat = 0) then
            rgrant(req)
         else
            locktbl[req^.id].rque.enq(req);
      end;     (* procedure dm_read *)


   entry procedure dm_write (req : topptr);
      begin
         await(time + 2);
         write(ddbug,'dm_write:  ');
         write(ddbug,req^.tid);
         write(ddbug,req^.id);
         write(ddbug,' ',req^.dest);
         writeln(ddbug,time);
         if(locktbl[req^.id].rstat = 0) and
            (locktbl[req^.id].wstat = 0) then
            wgrant(req)
         else  begin
            locktbl[req^.id].wque.enq(req);
            end;
      end;     (* procedure dm_write *)


   procedure dm_release (req : topptr);
      begin
      (*   write(ddbug,'dm_release:');
         write(ddbug,req^.typ);
```

```
            write(ddbug,req^.tid);
            write(ddbug,req^.id);
            write(ddbug,' ',req^.dest);
            writeln(ddbug,time);
         *)
            await(time + 2);
            if (req^.typ = 'w') then begin
               await(time + 2);
               locktbl[req^.id].wstat :=
                  locktbl[req^.id].wstat - 1;
               if (locktbl[req^.id].writeq  ) 0) then
                  wgrant(locktbl[req^.id].wque.deq);
               end
            else begin
               await(time + 2);
               locktbl[req^.id].rstat :=
                  locktbl[req^.id].rstat - 1;
               if (locktbl[req^.id].readq  ) 0) then
                  rgrant(locktbl[req^.id].rque.deq);
            end;
            req^.locked := false;
         end;    (* procedure dm_release *)


   procedure dm_unque(req : topptr);
      begin
         await(time + 2);
         if(req^.typ = 'w') then begin
            locktbl[req^.id].wque.unq(req);
            write(ddbug,'UNQ:  tid = ',req^.tid);
            writeln(ddbug,'objid = ',req^.id);
            end
         else begin
            write(ddbug,'UNQ:  tid = ',req^.tid);
            writeln(ddbug,'objid = ',req^.id);
            locktbl[req^.id].rque.unq(req);
            end;
      end;  (*procedure dm_unque*)


         entry procedure restart (xtion : transptr);
            begin
               write(ddbug,'restart transaction:  ');
               write(ddbug,xtion^.trid);
               writeln(ddbug,xtion^.nodeid);
               rscount := rscount + 1;
            end;    (* procedure restart *)

         entry procedure complete (xtion : transptr);
            begin
               stats.prt(xtion);
            end;



         entry procedure releaselocks  (tsact : transptr);
            var
               j : integer;
            begin
               writeln(ddbug,'--------RELEASELOCKS------tid= ',tsact^.trid)
               for j := 1 to tsact^.lockcount do begin
                  await(time + 2);
                  if(tsact^.transblk[j]^.locked) then
                     dm_release(tsact^.transblk[j])
                  else
                     dm_unque(tsact^.transblk[j]);
         end;
         await(time + 2);
         if (tsact^.restart) then
            restart(tsact)
         else
            complete(tsact);
      end;   (* procedure releaselocks *)


   entry process getlocks[50](tsact : transptr);
      begin
         writeln(ddbug,'---------------GETLOCKS------TID= ',tsact^.triu);
         commit[tsact^.trid] := true;
         await(time + 2);
         while (countlocks[tsact^.trid] < tsact^.lockcount)
            and(not tsact^.restart) do begin
            write(ddbug,'getlocks: countlocks = ');
            write(ddbug,tsact^.trid);
            writeln(ddbug,countlocks[tsact^.trid]);
            await(time + 2);
            link[tsact^.trid].recvlock(tsact);
            writeln(ddbug,'getlocks: recvlock--tid = ',tsact^.trid);
            end;
         commit[tsact^.trid] := false;
      end;   (* procedure getlocks *)



   procedure no_deadlock(tsact : transptr);
      var
         clock : integer;
      begin
         writeln(ddbug,'---------------NODEADLOCK-----TID = ',tsact^.trid)
         writeln(ddbug,'no=dead: time =', time,tsact^.trid);
         await(time + 4);
         writeln(ddbug,'no-dead2: time =', time,tsact^.trid);
         clock := time;
         while ((time-clock < timelimit) and (commit[tsact^.trid]))
            do begin
            writeln(ddbug,'do-dead--in loop time,clock, t-c =');
            writeln(ddbug,'no=dead :', time,clock,time-clock);
            await(time + 1);
            end;
         if (commit[tsact^.trid])
            then begin
               commit[tsact^.trid] := false;
               await(time + 3);
               tsact^.restart := true;
               link[tsact^.trid].sendlock;   (*unblock getlock*)
               write(ddbug,'no_dead: restart :');
               writeln(ddbug,tsact^.trid);

               end;
         writeln(ddbug,'call releaselocks tid = ',tsact^.trid);
         releaselocks(tsact);
         synch[tsact^.trid].send;
      end;   (* procedure no_deadlock *)

 entry process dm_r_w[125](tsact: transptr);
    var
       k : integer;
    begin
    writeln(ddbug,'----------DM_R_W-------TID = ',tsact^.trid);
       k := 0;
       while (k < tsact^.lockcount) do begin
          await(time + 3);
```

```
                    k := k + 1;
                    if (tsact^.transblk[k]^.typ = 'r') then
                       dm_read(tsact^.transblk[k])
                    else
                       dm_write(tsact^.transblk[k]);
                    end;
                 no_deadlock(tsact);
                 end;  (*process dm_r/w *)

  (*          entry procedure synchproc(tsact : transptr);
              begin
                 writeln(ddbug,'--------------SYNCHPROC----tid = ',tsact^.trid);
              end;
*)

        init;
           var
              i : integer;

           begin
           for i := 1 to n do begin
                 locktbl[i].rstat := 0;
                 locktbl[i].wstat := 0;
                 locktbl[i].readq := 0;
                 locktbl[i].writeq := 0;
                 locktbl[i].r_ts := 0;
                 locktbl[i].w_ts := 0;
                 end;
           end;

        end;   (*object dm_mgr*)


      entry process tm[3400] (tsact : transptr);
      label 1;
      var
         k : integer;
         tmm : object
            path 1:(phasetwo) end;

        entry procedure phasetwo (tsact : transptr);

           begin
              await(time + 4);
              countlocks[tsact^.trid] := 0;
              dm.dm_r_w(tsact);
              dm.getlocks(tsact);
              synch[tsact^.trid].recv(tsact);
           end;   (* process phase_two *)
        end;  (*object tmm *)

        function unirand : integer;    (*returns uniform(0,10)*)

            begin
               unirand := trunc(10*(ceed / 65535.0));
               ceed := (25173 * ceed + 13849) mod 65535;
            end;   (* unirand *)

      begin
         1: await(time + 2);
         tmm.phasetwo(tsact);
         write(ddbug,'for tid = ',tsact^.trid);
         if(tsact^.restart) then begin
            writeln(ddbug,')++++++++should goto 1 now+++++++++++');
            tsact^.restart := false;              (* reset flag *)
            goto 1;
         end;
   end;    (* process tm *)


   end; (* object trans_mgr *)

 (*************************************************************************)


(*trans_gen is the transaction generator located at each node.
  this simulates the processes at each node of the distributed
  dbms.  trans_gen delays for poisson(lamdal), then issues a new
  transaction.  the transaction is forwarded to the appropriate
  transaction manager for negotiation *)


   process trans_gen[33000] (nodeid : integer);

      var
         seed : integer;
         count : integer;
         trans : transptr;
         objid : integer;
         j : integer;
         start : integer;
         trand : real;


(* generate provides a number of random variable generators
that are used by trans_gen in generating transactions    *)

       gen : object
           path (1:(uniform) ,expo, poisson , geom)  end;




(* uniform generates random numbers with a uniform distribution between
a and b.  it modifies the global variable seed. *)
           entry function uniform  : real;

                begin
                   uniform := seed / 65535.0;
                   seed := (25173 * seed + 13849) mod 65535;
                end;  (* uniform *)


(* expo generates an exponentially distributed random varaible
   with mean lamda *)

             entry function expo(lamda:integer) : real;
                begin
                   expo := (-ln(1-uniform) * lamda);
                end;


(* poisson returns random numbers according to a poisson
distribution with mean = lambda.  since it uses uniform, seed
is modified. *)
           entry function poisson (lambda : integer) : integer;
                var
                   b , t : real;
                   gen   : integer;
                begin
                   b := exp(-lambda);
                gen := -1;
                t := 1.0;
                repeat
                    t := t + uniform;
                    gen := gen + 1
                until (t-b < 0.0);
                poisson := gen;
            end;   (* poisson *)


(*function geom samples from a geometric distribution with the
parameter prob.  prob is the probability of an event occuring
within a specific period of time *)

   entry function geom (prob : real) : integer;
          var
             i : integer;

          begin
             i := 0;
             while (uniform >= prob)  do begin
                i := i + 1;
                end;
             geom := i;
          end;   (* geom *)


      end; (* object gen *)


      procedure tr_gen_debugcc (txxt : transptr);
         var
            j : integer;
            i : integer;

         begin
            writeln(ddbug,'TRANSACTION ARRIVAL AT : ',time);
            writeln(ddbug,'     node id    = ',txxt^.nodeid);
            writeln(ddbug,'     tr_ts      = ',txxt^.tr_ts);
            writeln(ddbug,'     restart    = ',txxt^.restart);
            writeln(ddbug,'     lockcount  = ',txxt^.lockcount);
            writeln(ddbug,'     tr_id      = ',txxt^.trid);
            writeln(ddbug,'     destnode   = ',txxt^.destnode);
            writeln(ddbug,'     tread      = ',txxt^.tread);
            writeln(ddbug,'     twrite     = ',txxt^.twrite);
            i := 0;
            while (i < txxt^.lockcount) do begin
               i := i + 1;
               write(ddbug,txxt^.transblk[i]^.typ);
               writeln(ddbug,txxt^.transblk[i]^.id);
               write(ddbug,' ');
               end;

         end;   (* tr_gen_debugcc *)



      begin
         seed := sseed + 11*nodeid;
         count := 0;
         while count < runjobs  do begin
            count := count + 1;
            writeln(ddbug,'before delay time = ',time);
            await(time + trunc(gen.expo(lamda1)));
            writeln(ddbug,'after delay time = ',time);
            new(trans);
         trans^.nodeid := nodeid;
         trand := gen.uniform;        (* determine accessed noed*)
         writeln(ddbug,'call: trand = gen.uniform');
         if trand > 0.8 then
            trans^.destnode := (nodeid+1) mod (nodes+1);
         if trand > 0.9 then
            trans^.destnode := (nodeid+2) mod (nodes+1);
         if trand <= 0.8 then
            trans^.destnode := nodeid;

         trans^.tnum := count;
         trans^.trid := count*10 + nodeid;
         trans^.tr_ts := time;
         trans^.restart := false;
         trans^.tread := 0;
         trans^.twrite := 0;
         trans^.lockcount := gen.poisson(lamda2) + 1;;
         if (trans^.lockcount > tranmax) then
            trans^.lockcount := tranmax;    (* limit to tranmax requests*)
         objid := gen.geom(p)+1;
         writeln(ddbug,'call gen.geom');
         if (objid > n - trans^.lockcount) then
            objid := n - trans^.lockcount;
         j := 0;
         while (j < trans^.lockcount) do begin
            j := j + 1;
            if (gen.uniform < 0.3) then begin  (*write req^.est*)
               writeln(ddbug,'inside if(gen.uniform)');
               trans^.twrite := trans^.twrite + 1;
               new(trans^.transblk[j]);
               trans^.transblk[j]^.id := objid;
               trans^.transblk[j]^.locked := false;
               trans^.transblk[j]^.typ := 'w';
               trans^.transblk[j]^.dest := trans^.destnode;
               trans^.transblk[j]^.tid := trans^.trid;
               trans^.transblk[j]^.tnum := count;
               end
            else begin      (* read request *)
               trans^.tread := trans^.tread + 1;
               new(trans^.transblk[j]);
               trans^.transblk[j]^.id := objid;
               trans^.transblk[j]^.locked := false;
               trans^.transblk[j]^.typ := 'r';
               trans^.transblk[j]^.dest := trans^.destnode;
               trans^.transblk[j]^.tid := trans^.trid;
               trans^.transblk[j]^.tnum := count;
               end;
            objid := objid + 1;
            end;
         trans_mgr[trans^.destnode].tm(trans);
         tr_gen_debugcc(trans);


      end;
   end;    (* process trans_gen *)




process spawn[100000];
  var
    i : integer;

    begin   (* spawn *)
       ceed := sseed;
```

```
        for i := 0 to nodes do
           trans_gen(i);   (*send node id*)
     end;     (* spawn *)

(**************************************************************************)


  begin   (* main routine*)
       rewrite(ddbug);
       readln(lamdal);
       readln(runjobs);
       writeln(ddbug,'Lambal = ',lamdal);
       writeln(ddbug,'runjobs = ',runjobs);
       spawn;
       delay(10000);
  end.
```

APPENDIX B

BASIC TIMESTAMP ORDERING SIMULATOR

```
(**************************************************************************)
(*                        BASIC TIMESTAMP ORDERING                        *)
(**************************************************************************)

program concurrency (input,output,debug);


   const    (*************************************************************)
      tranmax = 40;           (* max number of variables accessed     *)
      n = 100;                (* number of dataobjects at each node   *)
      landa2 = 5;             (* mean of lockcount generator          *)
      wpct = 0.3;             (* percent of accesses that are writes *)
      p = 0.0957;             (* mean of geometric distribution       *)
      nodes = 2;              (* number of nodes(-1) in system        *)
      sseed = 12345;          (* initial seed for rand num generator *)



   type     (**************************************************************)
      dataobj = record          (* data object                 *)
         r_ts : integer;        (* last read timestamp         *)
         w_ts : integer;        (* last write timestamp        *)
         end;

      topptr = ^toper;
      toper = record            (* transaction operation       *)
         id : integer;          (* data object id              *)
         tid : integer;         (* transaction id              *)
         tnum : integer;        (* transaction number          *)
         typ : char;            (* type of access: write/read*)
         dest : integer;        (* access node id              *)
         ts : integer;          (* transaction timestamp       *)
         next : topptr;         (* pointer                     *)
         end;

      transptr = ^transrec;
      transrec = record               (* transaction record                      *)
         nodeid : integer;            (* source node id                          *)
         destnode : integer;          (* destination node id                     *)
         restart : boolean;           (* restart flag                            *)
         carry : integer;             (* sum of prior processing time            *)
         tr_ts : integer;             (* transaction timestamp                   *)
         lockcount : integer;         (* tot # of locks requested                *)
         trid : integer;              (* transaction id                          *)
         tnum : integer;              (* transaction number                      *)
         tread : integer;             (* # read locks requested                  *)
         twrite : integer;            (* # write locks                           *)
         transblk : array[1..tranmax] of topptr;(*array of operations *)
         end;




      (**************************************************************)
   var
      debug : text;                   (* debug output file *)
      runjobs : integer;              (* length of simulation: jobs per node *)
      landa1 : integer;               (* mean of arrival generator          *)
      rscount : integer;              (* # of restarted transactions *)
      stats : object
         path 1:(prt) end;
```

```
  var
     tput : real;             (* average throughput                *)
     rtime : real;            (* average response time             *)
     fincount : real;         (* # completed jobs                  *)
     trtime : real;           (* transaction response time         *)

  entry procedure prt(xxtion : transptr); (* print transaction stats*)
     begin
        fincount := fincount + 1.0;
        tput := fincount/time;
        trtime := xxtion^.carry + time - (xxtion^.tr_ts div 10);
        rtime := (((fincount-1)/fincount) * rtime)
                    + ((1/fincount)*trtime);
        writeln(debug, '---------------------------------');
        writeln(debug, 'transaction completed at time :  ', time);
        writeln(debug, 'total jobs completed = ', fincount);
        writeln(debug, 'tr_nodeid = ', xxtion^.nodeid);
        writeln(debug, 'tr_timestamp = ', xxtion^.tr_ts);
        writeln(debug, 'tr_id = ', xxtion^.trid);
        writeln(debug, 'TR response time = ', trtime);
        writeln(debug, 'ave response time = ', rtime);
        writeln(debug, 'ave throughput = ', tput);
        if(trunc(fincount) = (nodes+1) * runjobs) then begin
           writeln('Average Response Time = ', rtime);
           writeln('Average Throughput = ', tput);
           end;
     end;    (* procedure prt *)

   init;
     begin
        tput := 0.0;
        rtime := 0.0;
        fincount := 0.0;
        rscount := 0;
        trtime := 0.0;
     end;
   end; (*object stats *)

(* TRANSACTION MANAGER *)
trans_mgr : array[0..nodes] of object
path    tm  end;

var
   dbase : array[1..n] of dataobj;


(* dm is the data manager located at each node
this version of dmgr implements the basic 2pl
concurrency control algorithm*)

dm : object

path (dm_read,dm_write),1:(phaseone;phasetwo), 1:(restart,complete) end

type
   dataobj = record
      r_ts : integer;
      w_ts : integer;
      end;

entry procedure dm_read (req : topptr);
   begin
      await(time+1);
      write(debug, 'dm_read:   ');
      write(debug,req^.tid);
      write(debug,req^.id);
      write(debug,' ',req^.dest);
      writeln(debug,time);
      dbase[req^.id].r_ts := req^.ts;
   end;     (* procedure dm_read *)


entry procedure dm_write (req : topptr);
   begin
      await(time+1);
      write(debug,'dm_write: ');
      write(debug,req^.tid);
      write(debug,req^.id);
      write(debug,' ',req^.dest);
      writeln(debug,time);
      dbase[req^.id].w_ts := req^.ts;
   end;     (* procedure dm_write *)


      entry procedure restart (xtion : transptr);
         begin
            await(time+1);
            write(debug,'restart transaction: ');
            write(debug,xtion^.trid);
            writeln(debug,xtion^.nodeid);
            xtion^.carry := xtion^.carry + (time - xtion^.tr_ts div 10)
            rscount := rscount + 1;
         end;   (* procedure restart *)

      entry procedure complete (xtion : transptr);
         begin
            stats.prt(xtion);
         end;




      entry procedure phasetwo(tsact : transptr);
         var
            k : integer;
         begin
            writeln(debug,'-------PHASETWO-----tid = ',tsact^.trid);
            k := 0;
            await(time+2);
            while (k < tsact^.lockcount) and (not tsact^.restart)
               do begin
               await(time+3);
               k := k + 1;
               if(tsact^.transblk[k]^.typ = 'r') then
                  dm_read(tsact^.transblk[k])
               else
                  dm_write(tsact^.transblk[k]);
               end;
            await(time+2);
            if (tsact^.restart) then
               restart(tsact)
            else
               complete(tsact);
         end;   (* procedure getlocks *)



     entry procedure phaseone(tsact: transptr);
        var
```

```
          k : integer;
        begin
          writeln(debug,'---------------PHASEONE-----TID= ',tsact^.trid)
          k := 0;
          while (k < tsact^.lockcount) and (not tsact^.restart) do begin
            await(time+3);
            k := k + 1;
            if (tsact^.transblk[k]^.typ = 'r') then begin
              if(tsact^.transblk[k]^.ts <
                dbase[tsact^.transblk[k]^.id].r_ts)
                then tsact^.restart := true;
              end
            else if (tsact^.transblk[k]^.ts <
                    dbase[tsact^.transblk[k]^.id].w_ts) then
              tsact^.restart := true;
            end;
          end; (*process phaseone *)


    init;
      var
        i : integer;

      begin
        for i := 1 to n do begin
          dbase[i].r_ts := 0;
          dbase[i].w_ts := 0;
          end;
      end;

    end;  (*object dm_mgr*)


entry process tm[100] (tsact : transptr);
label 1;
var
    k : integer;
    tmm : object
      path 1:(twophase) end;



      entry procedure twophase (tsact : transptr);


      begin
        writeln(debug,'-------------TWOPHASE----for tid= ',tsact^.trid);
        await(time + 2);
        dm.phaseone(tsact);
        dm.phasetwo(tsact);
      end;  (* process phase_two *)
  end;  (* object tmm *)

 begin
    1: await(time+3);
    if(tsact^.restart) then begin
      await(time+3);
      tsact^.restart := false;
      tsact^.tr_ts := time;
      writeln(debug,'new tr_ts for tid = ',tsact^.trid,time);
      k := 0,
      while(k < tsact^.lockcount) do begin
        await(time+2);
        k := k + 1;
        tsact^.transblk[k]^.ts := tsact^.tr_ts;
        end;
```

```
            end;
         tmn.twophase(tsact);
         write(debug,'for tid = ',tsact^.trid);
         if(not tsact^.restart) then
            writeln(debug,'transcation completed');
         if(tsact^.restart) then begin
            writeln(debug,'!++++++++should goto 1 now++++++++++++');
            goto 1;
            end;
      end;   (* process tm *)


      end; (* object trans_mgr *)

   (**************************************************************************)


  (*trans_gen is the transaction generator located at each node.
    this simulates the processes at each node of the distributed
    dbms.  trans_gen delays for poisson(lamdal), then issues a new
    transaction.  the transaction is forwarded to the appropriate
    transaction manager for negotiation *)


      process trans_gen[9000] (nodeid : integer);

         var
            seed : integer;
            count : integer;
            trans : transptr;
            objid : integer;
            j : integer;
            start : integer;
            trand : real;


  (* generate provides a number of random variable generators
  that are used by trans_gen in generating transactions    *)

         gen : object
             path (1:(uniform),expo, poisson , geom)  end;



  (* uniform generates random numbers with a uniform distribution between
  a and b.  it modifies the global variable seed. *)
             entry function uniform  : real;

                 begin
                    uniform := seed / 65535.0;
                    seed := (25173 * seed + 13849) mod 65535;
                 end;   (* uniform *)


  (* expo returns an exponentially distributed random variable with
     mead landa *)

             entry function expo(lamda : integer) : real;
                 begin
                   expo := (-ln(1-uniform) * lamda);
                 end;


  (* poisson returns random numbers according to a poisson
distribution with mean = lambda.  since it uses uniform, seed
is modified. *)
            entry function poisson (lambda : integer) : integer;
                var
                    b , t : real;
                    gen   : integer;
                begin
                    b := exp(-lambda);
                    gen := -1;
                    t := 1.0;
                    repeat
                        t := t * uniform;
                        gen := gen + 1
                    until (t-b < 0.0);
                    poisson := gen;
                end;    (* poisson *)


(*function geom samples from a geometric distribution with the
parameter prob.  prob is the probability of an event occuring
within a specific period of time *)

    entry function geom (prob : real) : integer;
            var
                i : integer;

            begin
                i := 0;
                while (uniform >= prob)  do begin
                    i := i + 1;
                    end;
                geom := i;
            end;    (* geom *)


        end; (* object gen *)


        procedure tr_gen_debug (txxt : transptr);
            var
                j : integer;
                i : integer;

            begin
                writeln(debug,'TRANSACTION ARRIVAL AT :',time);
                writeln(debug,'     node id    = ',txxt^.nodeid);
                writeln(debug,'     tr_ts      = ',txxt^.tr_ts);
                writeln(debug,'     restart    = ',txxt^.restart);
                writeln(debug,'     lockcount  = ',txxt^.lockcount);
                writeln(debug,'     tr_id      = ',txxt^.trid);
                writeln(debug,'     destnode   = ',txxt^.destnode);
                writeln(debug,'     tread      = ',txxt^.tread);
                writeln(debug,'     twrite     = ',txxt^.twrite);
                i := 0;
                while (i < txxt^.lockcount) do begin
                    i := i + 1;
                    write(debug,txxt^.transblk[i]^.typ);
                    writeln(debug,txxt^.transblk[i]^.id);
                    write(debug,' ');
                    end;

            end;  (* tr_gen_debug *)


        begin
```

```
   seed := sseed + 11*nodeid;
   count := 0.
   while count < runjobs  do begin
       count := count + 1;
       await(time + trunc(gen.expo(lamda1)));
       new(trans);
       trans^.nodeid := nodeid;
       trand := gen.uniform;        (* determine accessed noed*)
       if trand  > 0.8 then
           trans^.destnode := (nodeid+1) mod (nodes+1);
       if trand  > 0.9 then
           trans^.destnode := (nodeid+2) mod (nodes+1);
       if trand <= 0.8 then
           trans^.destnode := nodeid;
       trans^.tnum := count;
       trans^.trid := count*10 + nodeid;
       trans^.tr_ts := time*10 + nodeid;
       trans^.restart := false;
       trans^.carry := 0;
       trans^.tread := 0;
       trans^.twrite := 0;
       trans^.lockcount := gen.poisson(lamda2) + 1;;
       if (trans^.lockcount  > tranmax) then
           trans^.lockcount := tranmax;    (* limit to tranmax requests*)
       objid := gen.geom(p)+1;
       if (objid  > n - trans^.lockcount) then
           objid := n - trans^.lockcount;
       j := 0;
       while (j < trans^.lockcount) do begin
           j := j + 1;
           if (gen.uniform < 0.3) then begin  (*write req^.est*)
               trans^.twrite := trans^.twrite + 1;
               new(trans^.transblk[j]);
               trans^.transblk[j]^.id := objid;
               trans^.transblk[j]^.ts := trans^.tr_ts;
               trans^.transblk[j]^.typ := 'w';
               trans^.transblk[j]^.dest := trans^.destnode;
               trans^.transblk[j]^.tid := trans^.trid;
               trans^.transblk[j]^.tnum := count;
               end
           else begin       (* read request *)
               trans^.tread := trans^.tread + 1;
               new(trans^.transblk[j]);
               trans^.transblk[j]^.id := objid;
               trans^.transblk[j]^.ts := trans^.tr_ts;
               trans^.transblk[j]^.typ := 'r';
               trans^.transblk[j]^.dest := trans^.destnode;
               trans^.transblk[j]^.tid := trans^.trid;
               trans^.transblk[j]^.tnum := count;
               end;
           objid := objid + 1;
           end;
        trans_mgr[trans^.destnode].tm(trans);
        tr_gen_debug(trans);


    end;
end;     (* process trans_gen *)
```

```
process spawn[100000];
  var
    i : integer;

    begin   (* spawn *)
       for i := 0 to nodes do
           trans_gen(i);   (*send node id*)
    end;     (* spawn *)

(****************************************************************************)


 begin  (* main routine*)
     rewrite(debug);
     readln(lamda1);
     readln(runjobs);                    TIMESTAMP ORDERING
     writeln('==.===-==-== ==BASIC ~~TWO PHASE LOCKING~~==============');
     writeln('Lamda 1 = ',lamda1);
     writeln('Runjobs = ',runjobs);

     spawn;
     delay(10000);
 end.
```

REFERENCES

[Bernstein and Goodman, 81] Bernstein, P.A. and Goodman, N., "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-219.

[Bray, 76] Bray, O.H., "Distributed Database Design Considerations," Tutorial Distributed Processing, IEEE Computer Society, 1981, pp. 465-471.

[Champine, 77] Champine, G.A., "Six Approaches to Distributed Databases," Datamation, May, 1977.

[Chu and Ohlmacher, 74] Chu, W.W. and Ohlmacher, G., "Avoiding Deadlock in Distributed Databases," Proceedings of the ACM-National Symposium, 1974.

[Date, 83] Date, C.J., An Introduction to Database Systems, Vol. II, Addison-Wesley (Reading, Mass: 1983).

[Garcia-Molina, 79] Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed System," Report No. STAN-CS-79-744, Department of Computer Science, Stanford University, June 1979.

[Kolstad and Campbell, 80] Kolstad, R.B. and Campbell, R.H., "Path Pascal User Manual," Department of Computer Science, University of Illinois at Champaign-Urabana, 1980.