International Journal of Modelling and Simulation Vol. 15, No. 2 1995

# A PERFORMANCE EVALUATION MODEL FOR DISTRIBUTED REAL-TIME DATABASES

Özgür Ulusoy* and Geneva G. Belford**

## Abstract

A real-time database system (RTDBS) is designed to provide timely response to the transactions of data-intensive applications. Each transaction processed in an RTDBS is associated with a timing constraint in the form of a deadline. Efficient transaction-scheduling algorithms are required to minimize the number of missed transaction deadlines. In this paper a performance evaluation model is provided to enable distributed RTDBS designers to analyze transaction scheduling algorithms. The model is developed progressing from a simple mathematical analysis to complicated simulations. The performance is expressed in terms of the fraction of satisfied transaction deadlines. The paper also provides an example simulation experiment implemented using the model presented.

## Key Words

Real-time database systems, transaction scheduling, performance evaluation

## 1. Introduction

A real-time database system (RTDBS) is a database system designed to provide real-time response to the transactions of data-intensive applications such as stock markets, computer-integrated manufacturing, telephone-switching, network management, and command and control systems. Each transaction submitted to an RTDBS is associated with a timing constraint in the form of a deadline. It is difficult in an RTDBS to meet all transaction deadlines due to the overhead of the consistency requirement for the underlying database. Conventional scheduling algorithms proposed to maintain data consistency are all based on transaction blocking and transaction restart, which makes it virtually impossible to predict computation times and hence to provide schedules that guarantee deadlines. The performance goal in RTDBS transaction scheduling is to minimize the number of transactions that miss their deadlines. A priority order is established among transactions based on their deadlines.

Distributed databases fit more naturally in the decentralized structures of many RTDB applications that are inherently distributed. Distributed RTDBSs provide shared data access capabilities to transactions; that is, a transaction is allowed to access data items stored at remote sites. While scheduling distributed RTDBS transactions, besides observing the timing constraints, one must also provide that the global consistency of the distributed database is preserved as well as the local consistency at each data site. To achieve this goal we require the exchange of messages carrying scheduling information between the data sites where the transaction is being executed. The communication delay introduced by message exchanges constitutes a substantial overhead for the response time of a distributed transaction. Thus, guaranteeing the response times of transactions (i.e., satisfying the timing constraints) is more difficult in a distributed RTDBS than in a single-site RTDBS.

The transaction scheduling problem in RTDBSs has been addressed by a number of recent studies ([1–6] in single-site systems and [7–10] in distributed systems); however, the emphasis in all these works is the development of new scheduling algorithms rather than provision of accurate performance models. The proposed algorithms have been evaluated using simple performance models, each with different simplifying assumptions. In this paper, we provide a detailed performance model to be used in the evaluation of distributed transaction scheduling algorithms in RTDBSs. The model is developed by progressing from a simple mathematical analysis to complicated simulations. The mathematical analysis is in terms of a probabilistic cost model to focus on the processing and IO "cost" (actual time requirements) of executing a distributed transaction in the system. The analysis enables us to determine the transaction load supported by the system, and to ensure that the values of system parameters are kept within reasonable ranges in simulation experiments.

The proposed model can be used in evaluating various components of transaction scheduling algorithms, among them the concurrency control protocol, CPU/disk scheduling algorithm, priority assignment policy, deadlock detection/recovery method, and transaction restart policy. The

An earlier version of this paper was published in the Proceedings of the IEEE 25th Annual Simulation Symposium, Orlando, Florida, April 1992.

* Department of Computer Engineering, Bilkent University, Bilkent, Ankara 06533 Turkey

** Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA

(paper no. 1993-27)

model enables users to study the relevant performance behavior of each component under diverse real-time and database environments to provide insights into different approaches. The performance is expressed in terms of the fraction of satisfied transaction deadlines.

Section 2 describes our performance model proposed for distributed RTDBSs. Section 3 presents the results of a performance experiment that evaluates some concurrency control protocols and priority assignment methods using the introduced performance model. Some final comments on the proposed model are provided in the last section.

## 2. A Distributed Real-Time Database System Model

The performance model is based on an open queuing model of a distributed database system that processes transactions associated with timing constraints. The model is driven by an external transaction source at a certain arrival rate.[1] We use a data distribution model that provides a partial replication of the distributed database. Each data item may have any number of replicas in the system (as many as the number of data sites). We believe that this model of data replication is a convenient one for real-time applications, because the number of copies of each data item might depend on the criticality and access frequency of the item. If a data item contains critical information and is required by many transactions from each data site, it must be fully replicated. On the other hand, there is no need to have many copies of a data item if it is not requested frequently.

Let $\{S_1, S_2, \ldots, S_n\}$ denote the set of $n$ data sites. The number of data items originating at each site is assumed to be the same. A data item $D$, originating at site $S_k$, $1 \leq k \leq n$, can have 0 to $n-1$ remote copies, each stored at a different site. Let $N(D)$ denote the number of replicas of $D$, including the one stored at the originating site. We assume that $N(D)$ can have any value from 1 to $n$ with equal likelihood, that is,

$$Prob(N(D) = i) = \frac{1}{n}.$$

Selecting the number of replicas of a data item from a uniform distribution was preferred to simplify various analyses performed in the rest of the paper.[2] It is also assumed that the copies of data item D are uniformly distributed over the remote data sites.

Mutual consistency of replicated data is achieved by using the "read-one, write-all" approach. A read operation on a data item can be performed on any copy of the data. If a local copy of the item exists, that copy is accessed without performing an intersite communication; otherwise, any of the remote copies is accessed with uniform probability. The write-all approach requires that a write operation on a data item be performed on all copies of the data.

Each site contains a transaction manager, a scheduler, a buffer manager, a resource manager, and a message server. The transaction manager is responsible for generating the workload for each data site. The arrivals at a site are assumed to be independent of the arrivals at the other sites. Each transaction submitted to the system is associated with a real-time constraint in the form of a deadline. The transaction is assigned a globally distinct real-time priority by using a specific priority assignment technique. Each transaction is executed to completion even if it misses its deadline. We model a distributed transaction as a master process that executes at the originating site of the transaction and a collection of cohorts that execute at various sites where the required data items reside. The transaction manager at a site is responsible for the creation of the master process and the cohort processes for each transaction submitted to that site. The cohorts are created dynamically as needed. There can be at most one cohort of a transaction at each site. For each operation executed, the transaction manager refers to the global data dictionary at its site to find out which data sites store copies of the data item referenced by the operation. Then the cohort(s) of the transaction at the relevant sites is activated to perform the operation. The master process coordinates of cohort processes; it does not itself perform any database operations. The priority of a transaction is carried by all its cohorts.

Atomic commitment of each transaction is provided by the centralized two-phase commit (2PC) protocol. An ideal commit protocol for RTDBSs should have the following desirable properties: predictability, low cost, fault tolerance, and consistency. However, to our knowledge no commit protocol with those properties has yet appeared in the literature. For applications with hard timing constraints, the consistency requirement can be relaxed to guarantee deadlines, but RTDBSs are basically restricted to soft timing constraints, and maintaining data consistency is the primary consideration in processing transactions. Some recent work has concentrated on the commitment problem for real-time transactions (e.g., [10, 12]). However, there is a tradeoff between the consistency and timeliness constraints in all proposed commitment approaches. We did not consider those protocols in our evaluations for several reasons. First, we did not want to sacrifice the data consistency. Second, those protocols are based on the hard deadlines assumption; that is, a transaction that missed its deadline is aborted. In our system, the deadlines are soft; a transaction continues to execute even if it misses its deadline. The protocol provided in [12]

[1] It is also possible to involve a closed queuing model in our analysis. In such a model, the transaction population in the system would be kept constant (i.e., there would be no external arrivals and each committed transaction would be restarted as a new transaction). We did not consider a closed-form solution as we believe that the behavior of a real RTDBS can be captured better by using an open-form solution.

[2] Another data distribution model was also considered in [11]. In that data model, each data item has exactly $N$ copies in the distributed system, where $1 \leq N \leq n$ (note that $N = 1$ and $N = n$ correspond to the no-replication and full-replication cases respectively). That model enabled us to execute the system at precisely specified levels of data replication.

is adaptive in the sense that under different situations, the system can dynamically change to a different commitment strategy. It is difficult to capture the behavior of an adaptive protocol in a mathematical analysis.

Returning to our model, each cohort of a transaction performs one or more database operations on specified data items. Concurrent data access requests of the cohort processes at a site are controlled by the scheduler at that site. The scheduler orders the data accesses based on the concurrency control protocol executed. The concurrency control protocol involves some form of priority-based decision policy in resolving data conflicts that may arise among the cohorts. Based on the real-time priority of a cohort, an access request of the cohort is either granted or results in blocking or aborting of the cohort. The scheduler at each site is also responsible for effecting aborts, when necessary, of the cohorts executing at its site.

If the access request of a cohort is granted, but the data item does not reside in main memory, the cohort waits until the buffer manager transfers the item from the disk into main memory. The FIFO page replacement strategy is used if no free memory space is available. Following the access, the data item is processed. When a cohort completes all its accesses and processing requirements, it waits for the master process to initiate 2PC. Following the successful commitment of the distributed transaction, the cohort writes its updates, if any, into the local database.

Each site's resource manager is responsible for providing IO service for reading/updating data items, and CPU service for processing data items, performing various concurrency control operations (e.g., conflict check, locking), and processing communication messages. Both CPU and IO queues are organized on the basis of the cohorts' real-time priorities. Preemptive-resume priority scheduling is used by the CPUs at each site; a higher priority process preempts a lower priority process, and the lower priority process can resume when there exists no higher priority process waiting for the CPU. Communication messages are given higher priority at the CPU than other processing requests.

There is no globally shared memory in the system, and all sites communicate via message exchanges over the communication network. A message server at each site is responsible for sending/receiving messages to/from other sites. It listens on a well-known port, waiting for remote messages.

Reliability and recovery issues were not addressed in this paper. We assumed a reliable system, in which no site failures or communication network failures occur.[3] It was also assumed that the network has enough capacity to carry any number of messages at a given time, and each message is delivered within a finite amount of time.

The set of parameters described in table 1 is used to specify the system configuration and workload. All sites of the system are assumed to be identical and to operate under the same parameter values. Each site is assumed to have one CPU and one disk. These simplifying assumptions aim to keep the analysis tractable.

[3] [11] provides an extension to the performance model to consider site failures.

Table 1
Distributed System Model Parameters

| | |
|---|---|
| *n* | Number of data sites in the distributed system |
| *local_db_size* | Size of database originated at each site |
| *mem_size* | Size of main memory at each site |
| *cpu_time* | CPU time to process a data item |
| *io_time* | IO time to access a disk-resident data item |
| *comm_delay* | Delay of a communication message between any two data sites |
| *mes_proc_time* | CPU time to process a communication message |
| *pri_assign_cost* | Processing cost of priority assignment |
| *lookup_cost* | Processing cost of locating a data item |
| *iat* | Mean interarrival time of transactions at a site |
| *tr_type_prob* | Fraction of transactions that are updates |
| *access_mean* | Mean number of data items accessed by a transaction |
| *data_update_prob* | Fraction of updated data items by an update transaction |
| *slack_rate* | Average slack time/processing time for a transaction |

The times between transaction arrivals at each site are exponentially distributed with mean iat. The transaction workload consists of both query and update transactions. The type of a transaction (i.e., query or update) is determined on a random basis using the parameter *tr_type_prob*, which specifies the update type probability. *access_mean* specifies the mean number of data items to be accessed by a transaction. Accesses are uniformly distributed among the data sites. Each transaction's deadline is set in proportion to the number of data items in its access list. The parameter *slack_rate* is used in determining the slack time of a transaction while assigning a deadline to the transaction. The deadline of a transaction is determined by the following formula:

deadline =
start-time + processing-time-estimate + slack-time

The value of slack time is chosen from the exponential distribution with a mean of *slack_rate* times processing-time-estimate. The processing-time-estimate of a transaction is calculated based on CPU/IO delay of the operations performed for the transaction [11].

### 2.1 Expected IO and CPU Utilizations at Each Data Site

A simple probabilistic cost model is used to determine the CPU and IO cost (in terms of expended time) of processing a distributed transaction in the system. The model is based on an evaluation of expected CPU and IO utilizations in terms of system parameters. It enables users to keep the system parameter values in appropriate ranges to obtain a desired level of resource utilization.

Assuming that the mean interarrival time between successive transaction arrivals at each data site is the same and is specified by the parameter *iat*, the expected IO utilization experienced at a site can be represented by the following formula:

$$U_{IO} = \frac{1}{iat} * (io\_cost_{l\_xact} + (n-1) * io\_cost_{r\_xact})$$

$io\_cost_{l\_xact}$ and $io\_cost_{r\_xact}$ denote the average IO cost (delay in mseconds) of a local and remote transaction respectively. The following subsection provides the specification of these variables in terms of the system parameters.

Similarly, the expected CPU utilization at a site will be:

$$U_{CPU} = \frac{1}{iat} * (cpu\_cost_{l\_xact} + (n-1) * cpu\_cost_{r\_xact})$$

$cpu\_cost_{l\_xact}$ and $cpu\_cost_{r\_xact}$ are the average CPU cost (execution time) of a local and remote transaction respectively.

*2.1.1 Average IO Cost of a Distributed Transaction at a Site*

Let us specify the average IO cost of transaction $T$ at a representative site $k$. We have two different cases based on the classification of the transactions executed at a site.

*Case 1: T is a local transaction (it originated at site k).*

The average IO cost of local transaction T at site k can be specified as:

$$io\_cost_{l\_xact} = access\_mean * io\_cost_{l\_op}$$

*access_mean* specifies the mean number of data items accessed by each transaction. $io\_cost_{l\_op}$ is the average IO cost of an operation submitted by $T$, and can be formulated as follows:

$$io\_cost_{l\_op} = P_r * P_L * t_r + P_w * P_L * (t_r + t_w)$$

$P_r(P_w)$ is the probability that the operation is a read (write). $t_r(t_w)$ specifies the average IO cost of executing a read (write) operation. It is assumed that each data item to be written is read first. $P_L$ is the probability that the data item accessed by the operation has a local copy (at site $k$). The values of these variables in terms of system parameters are provided in the Appendix.

*Case 2: T originated at a remote site.*

The average IO cost of remote transaction $T$ at site $k$ is:

$$io\_cost_{r\_xact} = access\_mean * io\_cost_{rem\_op}$$

$io\_cost_{rem\_op}$ is the average IO cost of an operation of $T$ at site $k$, and it can be specified by the following formula:

$$io\_cost_{rem\_op} = Pr * P_{read\_k} * t_r + P_w * P_{write\_k} * (t_r + t_w)$$

$P_{read_k}(P_{write_k})$ is the probability that remote transaction $T$ accesses a data item at site $k$ to perform the read (write) operation. These variables are formulated in the Appendix.

*2.1.2 Average Processing (CPU) Cost of a Distributed Transaction at a Site*

In formulating the average processing cost of a transaction $T$ at a site $k$, we again have to consider the two different transaction classes $T$ can belong to.

*Case 1: T is a local transaction.*

The average processing cost formula of local transaction $T$ is made up of three cost components corresponding to different stages of the transaction execution:

$$cpu\_cost_{l\_xact} = init\_cost_{l\_xact} + op\_proc\_cost_{l\_xact} + commit\_cost_{l\_xact}$$

The cost components specify the average processing cost of transaction initialization, operation execution, and atomic commitment respectively. Each of these components is formulated as follows:

$$init\_cost_{l\_xact} = pri\_assign\_cost + data\_locate\_cost + cohort\_init\_cost$$

*pri_assign_cost* is the parameter specifying the processing cost of priority assignment. *data_locate_cost* specifies the cost of locating the sites of data items in the access list of a local transaction. *cohort_init_cost* is the cost of initiating cohorts of a local transaction at remote sites.

$$data\_locate\_cost = access\_mean * lookup\_cost$$

Note that *lookup_cost* refers to the processing time to locate a single data item.

$$cohort\_init\_cost = (n-1) * P_{submit_j} * mes\_proc\_time$$

There exist $n-1$ remote data sites. $P_{submit_j}$ (see Appendix) is the probability that transaction $T$ submits at least one operation to remote site $j$. *mes_proc_time* specifies the CPU time required to process a communication message before being sent or after being received.

$$op\_proc\_cost_{l_xact} = op\_act\_cost + access\_mean * exec\_cost_{l\_op}$$

*op_act_cost* represents the CPU cost of activating $T$'s operations at remote sites (if remote data access is needed) and processing the corresponding "operation complete" messages sent back at the end of each operation execution. $exec\_cost_{l\_op}$ specifies the cost of processing each local operation.

$$op\_act\_cost = N_{op-submit} * 2 * mes\_proc\_time$$

$N_{op-submit}$, the average number of operations a local transaction submits to remote sites, is derived in the Appendix.

$$exec\_cost_{l\_op} = P_L * cpu\_time$$

Recall that $P_L$ is the probability that the data item accessed by the operation has a local copy.

$$commit\_cost_{l\_xact} = (n-1) * P_{submit_j} * (2 * mes\_proc\_time + mes\_proc\_time)$$

$commit\_cost_{l\_xact}$ is due to executing the 2PC protocol[4] for a committing local transaction. Recall that $P_{submit_j}$ is the probability that a local transaction submits operation to a remote site (e.g., site $j$). $2 * mes\_proc\_time$ corresponds to the cost of phase 1 of 2PC (sending a message to each of the cohort sites and processing the messages sent back from those sites), and the second term in the parentheses, that is, $mes\_proc\_time$, corresponds to phase 2 (sending the final decision message to cohort sites).

*Case 2: T originated at a remote site.*

Similar to the local transaction case, the average processing cost of remote transaction T can be specified as follows:

$$cpu\_cost_{r\_xact} = init\_cost_{r\_xact} + op\_proc\_cost_{r\_xact} + commit\_cost_{r\_xact}$$

$init\_cost_{r\_xact}$ specifies the average cost of processing the "initiate cohort" message for each remote transaction.

$$init\_cost_{r\_xact} = P_{op-from_j} * mes\_proc\_time$$

$P_{op-from_j}$ is the probability that remote transaction $T$ (from site $j$) submits a cohort to this site (to site $k$). This probability is equal to $P_{submit_j}$, derived in the Appendix.

$$op\_proc\_cost_{r\_xact} = access\_mean * (act\_msg\_cost + exec\_cost_{rem\_op})$$

$act\_msg\_cost$ specifies the average CPU cost for each of $T$'s operations due to processing "activate operation" message from T's site and sending back "operation complete" message at the end of operation execution. $exec\_cost_{rem\_op}$ represents the average CPU cost due to processing each operation of $T$.

$$act\_msg\_cost = (P_r * P_{read_k} + P_w * P_{write_k}) * (2 * mes\_proc\_time)$$

As noted earlier, $P_r(P_w)$ is the probability that the operation is a read (write), and $P_{read_k}(P_{write_k})$ is the probability that $T$ accesses a data item at this site (site $k$) to perform the read (write) operation.

$$exec\_cost_{rem\_op} = (P_r * P_{read_k} + P_w * P_{write_k}) * cpu\_time$$

$commit\_cost_{r\_xact}$ is the average processing cost due to executing the commit protocol[5] for remote transaction $T$.

$$commit\_cost_{r\_xact} = P_{op-from_j} * (2 * mes\_proc\_time + mes\_proc\_time)$$

$2 * mes\_proc\_time$ corresponds to the overhead of phase 1 and $mes\_proc\_time$ corresponds to the cost of phase 2 of 2PC protocol for each committing remote transaction.

## 3. A Performance Evaluation Example

A program to simulate our distributed system model was written in CSIM [13], which is a simulation language based on the C programming language. For each simulation experiment, the final results were evaluated as averages over 100 independent runs. Each configuration was executed for 500 transactions originated at each site. Ninety percent confidence intervals were obtained for the performance results. The width of the confidence interval of each data point is within 4% of the point estimate. The mean values of the performance results were used as final estimates.

Two important components of an RTDBS transaction scheduler were involved in our evaluation process: *Concurrency Control Protocol* and *Priority Assignment Policy.*

### 3.1 Concurrency Control Protocols

The concurrency control protocols studied are all based on the two-phase locking (2PL) method. In our earlier performance works ([6, 11]) we compared various lock-based, optimistic, and timestamp-ordering protocols and observed that lock-based protocols perform consistently better than the other ones under different system configurations and workloads. The optimistic and timestamp-ordering protocols performed well only under light transaction loads or when the data/resource contention in the system was low.

For the locking protocols studied, each scheduler manages locks for the data items stored at its site. Each cohort process executing at a data site has to obtain a shared lock on each data item it reads, and an exclusive lock on each data item it updates. In order to provide global serializability, the locks held by the cohorts of a transaction are maintained until the transaction has been committed. The concurrency control protocols differ in how the real-time priorities of the transactions are involved in scheduling the lock requests.

Local deadlocks are detected by maintaining a local Wait-For Graph (WFG) at each site. WFGs contain the "wait-for" relationships among the transactions. For the detection of global deadlocks a global WFG is used, which is constructed by merging local WFGs. One of the sites is employed for periodic detection/recovery of global deadlocks. A deadlock is recovered from by selecting the low-

[4] As the coordinator.

[5] As the participant.

est priority cohort in the deadlock cycle as a victim to be aborted. The master process of the victim cohort is notified to abort and later restart the whole transaction.

*Priority Inheritance Protocol (PI):* The priority inheritance method, proposed in [7], makes sure that when a transaction blocks higher priority transactions, it is executed at the highest priority of the blocked transactions; in other words, it inherits the highest priority. The idea is to reduce the blocking times of high-priority transactions.

In our distributed system model, when a cohort is blocked by a lower priority cohort, the latter inherits the priority of the former. Whenever a cohort of a transaction inherits a priority, the scheduler at the cohort's site notifies the transaction's master process by sending a priority inheritance message, which contains the inherited priority. The master process then propagates this message to the sites of other cohorts belonging to the same transaction, so that the priority of the cohorts can be adjusted.

*High Priority Protocol (HP):* This protocol resolves data conflicts in favor of high-priority transactions [1]. At the time of a data lock conflict, if the lock-holding cohort has higher priority than the cohort requesting the lock, the latter cohort is blocked. Otherwise, the lock-holding cohort is aborted and the lock is granted to the high-priority lock-requesting cohort. Upon the abort of a cohort, a message is sent to the master process of the aborted cohort to restart the whole transaction. The master process notifies the schedulers at all relevant sites to cause the cohorts belonging to that transaction to abort. It then waits for the abort confirmation message from each of those sites. When all the abort confirmation messages are received, the master can restart the transaction.

As a high-priority transaction is never blocked by a lower priority transaction, this protocol is deadlock-free.[6]

*No Priority Protocol (NP):* This protocol resolves lock conflicts by blocking a cohort that requests a lock that is already held. The cohort remains blocked until the conflicting lock is released. The real-time priority of the cohorts is not considered in processing the lock requests. Inclusion of this protocol aims to provide a basis of comparison for studying the performance of the priority-based protocols.

### 3.2 Priority Assignment Policies

*Earliest Deadline First (EDF):* A transaction with an earlier deadline has higher priority than a transaction with a later deadline. If any two of the transactions have the same deadline, the one that has arrived at the system earlier is assigned a higher priority (i.e., First Come First Served [FCFS] policy).

*Least Slack First (LSF):* The slack time of a transaction can be defined as the maximum length of time the transaction can be delayed and still satisfy its deadline. Between any two transactions, the one with less slack time is assigned higher priority by the LSF policy.

[6] The assumption here is that the real-time priority of a transaction does not change during its lifetime and that no two transactions have the same priority.

For this experiment, the static version of the LSF policy was implemented. This policy assigns the priority of a transaction based on its slack time when it is submitted to the system as a new transaction or when it is restarted. (The dynamic version of this policy evaluates the transaction priorities continuously. We did not implement it because of the considerable overhead incurred by continuous calculation of the priorities whenever needed.) The LSF policy assumes that each transaction provides its processing time estimate. Similar to the EDF policy, the tie breaker in the case of equal slack times is the FCFS policy.

*Random Priority Assignment (RPA):* Each transaction obtains a priority on a random basis. The priority assigned to a transaction is independent of that transaction's deadline.

Table 2 specifies the default values of the system parameters used in the simulation experiment. The values of *cpu_time* and *io_time* were chosen to yield a system with almost identical CPU and IO utilizations. Neither a CPU-bound nor an IO-bound system is intended. (It would be possible to have a CPU-bound or an IO-bound system by simply having different settings of the parameters *cpu_time* and *io_time*.) The experiment was conducted to evaluate the performance of various concurrency control protocols and priority assignment methods under different levels of transaction load. Mean time between successive transaction arrivals at a site (i.e., *iat*) was varied from 300 to 460 mseconds in steps of 40. This range of iat values corresponds to a CPU utilization of .94 to .61, and IO utilization of .91 to .60 at each data site. The performance metric used in the experiment is *success-ratio*; that is, the fraction of transactions that satisfy their deadlines.

Table 2
Distributed RTDBS Model Parameter Values

| | | | |
|---|---|---|---|
| *n* | 10 | *pri_assign_cost* (msec) | 1 |
| *local_db_size* | 200 | *lookup_cost* (msec) | 1 |
| *mem_size* | 500 | *iat* (msec) | 400 |
| *cpu_time* (msec) | 8 | *tr_type_prob* | .50 |
| *io_time* (msec) | 18 | *access_mean* | 6 |
| *comm_delay* (msec) | 5 | *data_update_prob* | .50 |
| *mes_proc_time* (msec) | 2 | *slack_rate* | 5 |

### 3.3 Simulation Results

Figs. 1, 2, and 3 display comparative real-time performance results of the concurrency control protocols for priority assignment policies EDF, LSF, and RPA respectively. To check the reliability of the results obtained, we calculated the variances and generated confidence intervals of data points. Table 3 provides the variance and the width of confidence interval obtained for each data point displayed in fig. 1. The mean values of 100 independent runs were used as final estimates. The width of the confidence in-

terval of each data point is always within 4% of the point estimate. Confidence intervals obtained with the other performance results (i.e., those displayed in figs. 2 and 3) also verified this observation.

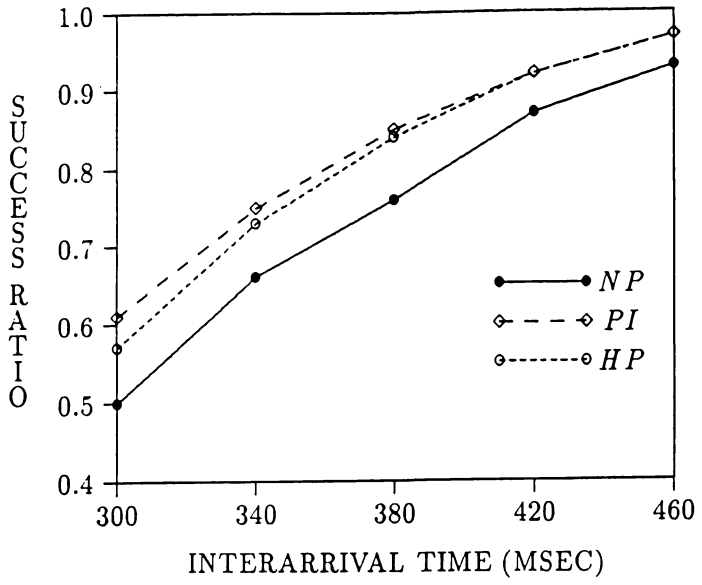


Figure 1. *Success-ratio* versus *iat* with EDF priority assignment policy.

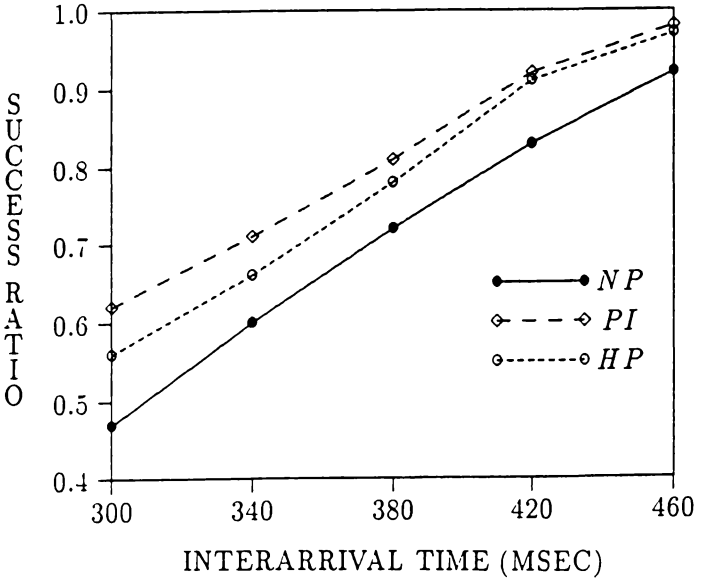


Figure 2. *Success-ratio* versus *iat* with LSF priority assignment policy.

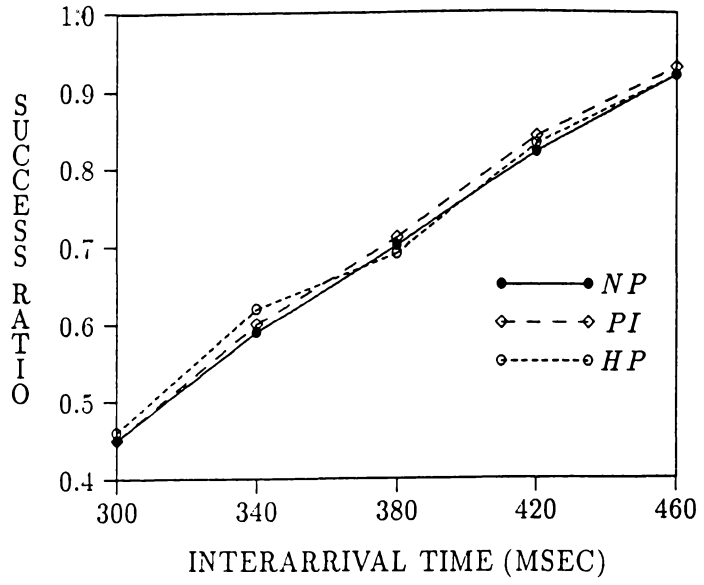


Figure 3. *Success-ratio* versus *iat* with random priority assignment policy.

Table 3
Mean ($\mu$), Variance ($\sigma^2$), and Width of Confidence Interval Obtained with Each Data Point in Fig. 1

| NP | | | PI | | | HP | | |
|---|---|---|---|---|---|---|---|---|
| $\mu$ | $\sigma^2$ | $\|CI\|$ | $\mu$ | $\sigma^2$ | $\|CI\|$ | $\mu$ | $\sigma^2$ | $\|CI\|$ |
| .50 | .00316 | .0185 | .61 | .00270 | .0171 | .57 | .00347 | .0194 |
| .66 | .00330 | .0189 | .75 | .00351 | .0195 | .73 | .00504 | .0234 |
| .76 | .00653 | .0266 | .85 | .00727 | .0280 | .84 | .00892 | .0311 |
| .87 | .00672 | .0270 | .92 | .00958 | .0322 | .92 | .00529 | .0239 |
| .93 | .00979 | .0325 | .97 | .01255 | .0369 | .97 | .00782 | .0291 |

Protocol NP exhibits the worst performance under varying levels of transaction load with priority assignment methods EDF and LSF. Furthermore, its performance is not affected by a change in the priority assignment policy implemented. These results are not surprising, as protocol NP does not include real-time priorities of transactions in data access scheduling decisions. Concurrency control protocols PI and HP both provide a considerable improvement in real-time performance over protocol NP when the EDF or LSF method is used in assigning priorities.

The performance of protocol PI is somewhat better, in general, than that of protocol HP for a wide range of mean interarrival time. Remember that protocol HP aborts low-priority transactions whenever necessary to resolve data conflicts. The overhead of transaction aborts in a replicated database system leads to the performance difference as against protocol HP. Aborting a transaction that has already performed some write operations causes considerable waste of IO/CPU resources at all the sites storing the copies of updated data. Especially under high transaction loads, protocol PI is preferable to protocol HP.

Comparing the performance results obtained with EDF and LSF priority assignment policies, we observe a small difference in the performance of the concurrency control protocol PI. The LSF policy results in an increase in the number of satisfied deadlines for this protocol. Because the restart ratio in this protocol is quite low,[7] most of the transactions keep the static priority assigned at the beginning of their execution. As a result, as is not the case in the EDF policy, the priority of a tardy transaction is not always higher than the priority of a nontardy transaction. Thus, a nontardy transaction can get a better chance to satisfy its deadline. The difference between the performance obtained with the two priority assignment policies is not striking for protocol HP, which employs transaction restart as well as transaction blocking in resolving data conflicts. Because the LSF policy updates priority of a transaction each time it is restarted, tardy transactions with adjusted priority are more likely to be scheduled before the nontardy transactions, which is always the case with the EDF policy.

With RPA, the performance of concurrency control protocols PI and HP is worse than that of the other priority assignment methods. Furthermore, their performance characteristics are not distinguishable from what is obtained with the NP concurrency control protocol. As the timing constraints of the transactions are not considered in determining the transaction priorities, they have no effect on the data access scheduling decisions of the protocols PI and HP. Thus, these two protocols cannot benefit from involving transaction priorities in access scheduling.

The conclusion that can be drawn from the performance experiment described in this section is that, under the parameter ranges explored, the best combination of concurrency control protocol and priority assignment technique seems to be the Priority Inheritance protocol with the Least Slack First policy.

## 4. Discussion

A performance evaluation model was proposed that enables users to analyze distributed transaction scheduling algorithms in RTDBSs. The performance model was developed in two steps. First, a mathematical analysis was performed to focus on the processing and IO cost of executing a distributed transaction in the system. The second step was the implementation of the distributed system model by using simulation. The ranges of parameter values used in the simulations were guided by the mathematical analysis to provide a high level of utilization of CPU and IO resources at each data site.

In the performance experiment presented here as an example, our RTDBS model was used to evaluate the performance of some concurrency control protocols and priority assignment methods. The performance results were provided in terms of the rate of transactions satisfying their deadlines. In addition to the concurrency control and priority assignment, the other components of a transaction scheduling algorithm (e.g., the CPU/disk scheduling algorithm, the deadlock detection/recovery method, the transaction restart policy) can be studied by using the proposed model. The example experiment investigated only the effects of transaction load on real-time performance. It is also possible to involve each of the other system parameters in the evaluation process.

[7] Blocking deadlock is the only source of transaction restarts in protocol PI.

## References


[1] R. Abbott & H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th Int. Conf. on Very Large Data Bases*, Los Angeles, CA, 1988, 1–12.

[2] R. Abbott & H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proc. 15th Int. Conf. on Very Large Data Bases*, Amsterdam, 1989, 385–396.

[3] J.R. Haritsa, M.J. Carey, & M. Livny, "On Being Optimistic About Real-Time Constraints," *ACM SIGACT-SIGMOD-SIGART*, Nashville, TN, 1990, 331–343.

[4] J.R. Haritsa, M.J. Carey, & M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proc. 11th Real-Time Systems Symposium*, Orlando, FL, 1990, 94–103.

[5] J. Huang, J.A. Stankovic, D. Towsley, & K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proc. 10th Real-Time Systems Symposium*, 1989, 144–153.

[6] Ö. Ulusoy & G.G. Belford, "Real-Time Transaction Scheduling in Database Systems," *Information Systems, 18*(8), 1993, 559–580.

[7] L. Sha, R. Rajkumar, & J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record, 17*, 1988, 82–98.

[8] L. Sha, R. Rajkumar, S.H. Son, & C.H. Chang, "A Real-Time Locking Protocol," *IEEE Trans. on Computers, 40*(7), 793–800.

[9] S.H. Son & C.H. Chang, "Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment," *Proc. 10th Int. Conf. on Distributed Computing Systems*, 1990, 124–131.

[10] Ö. Ulusoy & G.G. Belford, "Real-Time Lock Based Concurrency Control in a Distributed Database System," *Proc. 12th Int. Conf. on Distributed Computing Systems*, Yokohama, 1992, 136–143.

[11] Ö. Ulusoy, *Concurrency Control in Real-Time Database Systems*, Technical Report UIUCDCS-R-92-1762, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[12] N. Soparkar, E. Levy, H.F. Korth, & A. Silberschatz, *Adaptive Commitment for Real-Time Distributed Transactions*, Technical Report TR-92-15, Computer Science Department, University of Texas at Austin, 1992.

[13] H. Schwetman, "CSIM: A C-Based, Process-Oriented Simulation Language," *Proc. Winter Simulation Conference*, 1986, 387–396.


## Appendix: Probabilistic Variables

$P_r$: *Probability that an operation is a read.*

$$P_r = 1 - tr\_type\_prob * data\_update\_prob \quad (1)$$

$P_w$: *Probability that an operation is a write.*

$$P_w = tr\_type\_prob * data\_update\_prob \quad (2)$$

$t_r$: *Average IO cost of reading a data item.*

$$t_r = \begin{cases} 0 & \text{if } db\_size \leq mem\_size \\ (1 - \frac{mem\_size}{db\_size}) * io\_time & \text{otherwise} \end{cases}$$

*db_size* specifies the average size of database stored at a site. Based on our data replication model presented in Section 2, the value of *db_size* was derived in [11] in terms of parameter *local_db_size* as:

$$db\_size = (\frac{n+1}{2}) * local\_db\_size$$

$t_w$: *Average IO cost of writing a data item on the disk.*

$$t_w = io\_time$$

$P_L$: *Probability that a data item D accessed by an operation has a local copy.*

$$P_L = \sum_{i=1}^{n} P_i * P_{L|i}$$

$P_i$: Probability that the data item has $i$ copies in the system.
$P_{L|i}$: Probability that one of the $i$ copies of the item is local.

Assumption (Section 2):

$$P_i = \frac{1}{n} \quad (3)$$

$$P_{L|i} = \frac{\binom{n-1}{i-1}}{\binom{n}{i}} = \frac{i}{n} \quad (4)$$

$$P_L = \sum_{i=1}^{n} \frac{1}{n}(\frac{i}{n}) = \frac{n+1}{2n} \quad (5)$$

$P_{read\_k}$: *Probability that a read operation of a transaction originated at a remote site is performed at site k.*

$$P_{read\_k} = \sum_{i=1}^{n-1} P_{i,\bar{L}} * P_{k|i,\bar{L}} * P_{sel_k|i,\bar{L},k} \quad (6)$$

$P_i, \bar{L}$: Probability that the data item accessed by the operation has $i$ copies and none of the copies is local.
$P_{k|i,\bar{L}}$: Probability that site $k$ stores a copy of the data item given that none of the $i$ copies of the item is local.
$P_{sel_k|i,\bar{L},k}$: Probability that the item copy stored at site $k$ is selected to be read, given that among $i$ copies of the item there is no local copy while site $k$ has one.

$$P_{i,\bar{L}} = P_i * P_{\bar{L}|i}$$

$P_i$: Probability that the data item has $i$ copies in the system (see (3)).
$P_{\bar{L}|i}$: Probability that none of the $i$ copies of the item is local.

Using (4),

$$P_{\bar{L}|i} = 1 - \frac{i}{n}$$

$$P_{i,\bar{L}} = \frac{1}{n})1 - \frac{i}{n}) \quad (7)$$

$$P_{k|i,\bar{L}} = \frac{\binom{n-2}{i-1}}{\binom{n-1}{i}} = \frac{i}{n-1} \quad (8)$$

$$P_{sel_k|i,\bar{L},k} = \frac{1}{i} \quad (9)$$

Combining (6), (7), (8), and (9), we obtain

$$P_{read_k} = \sum_{i=1}^{n-1} \frac{1}{n}(1 - \frac{i}{n})(\frac{i}{n-1})\frac{1}{i} = \frac{1}{2n} \quad (10)$$

$P_{write_k}$: *Probability that site k stores a copy of the data item to be accessed by a write operation of a transaction originated at a remote site.*

$P_{write_k} = P_L$, because the probability of finding a copy of a data item at the local site or at a remote site is the same due to the uniform distribution of data over the sites.

Thus, using (5),

$$P_{write_k} = \frac{n+1}{2n} \quad (11)$$

$P_{submit_j}$: *Probability that a transaction originating at site k submits at least one operation to site j.*

$$P_{submit_j} = 1 - P_{submit_{\bar{j}}}$$

$P_{submit_{\bar{j}}}$: Probability that no operation is submitted to site $j$.

$$P_{submit_j} = \prod_{i=1}^{access\_mean} P_{i,submit_{\bar{j}}} \quad (12)$$

$P_{i,submit_{\bar{j}}}$: Probability that the $i$th operation of the transaction does not access site $j$.

$$P_{i,submit_{\bar{j}}} =$$

$$P_r * P_{read_{\bar{j}}} + P_w * P_{write_{\bar{j}}} (i = 1, 2, \ldots, access\_mean) \quad (13)$$

$P_r$: Probability that the operation is a read (see (1)).
$P_{read_{\bar{j}}}$: Probability that the data is not read from site $j$.
$P_w$: Probability that the operation is a write (see (2)).
$P_{write_{\bar{j}}}$: Probability that the data is not written at site $j$.

$$P_{read_{\bar{j}}} = 1 - P_{read_j}$$

$P_{read_j}$: Probability that the data item is read from site $j$; from (10):

$$P_{read_j} = \frac{1}{2n}$$

$$P_{read_j} = 1 - \frac{1}{2n} \tag{14}$$

$$P_{write_j} = 1 - P_{write_j}$$

$P_{write_j}$: Probability that site $j$ stores a copy of the data item that is going to be accessed by the write operation; from (11):

$$P_{write_j} = \frac{n+1}{2n} \tag{15}$$

$$P_{write_j} = 1 - \frac{n+1}{2n} \tag{16}$$

Using (13), (1), (14), (2), and (16),

$$P_{i,submit_j} = (1-\frac{1}{2n}) - \frac{1}{2} * tr\_type\_prob * data\_update\_prob \tag{17}$$

Substituting (17) into (12),

$$P_{submit_j} = [P_{i,submit_j}]^{access\_mean}$$

$$= [(1-\frac{1}{2n}) - \frac{1}{2} * tr\_type\_prob * data\_update\_prob]^{access\_mean}$$

$$P_{submit_j} =$$

$$1-[(1-\frac{1}{2n}) - \frac{1}{2} * tr\_type\_prob * data\_update\_prob]^{access\_mean}$$

$N_{rem-copy}$: *Average number of remote copies of each data item accessed.*

$$N_{rem-copy} = (n-1) * P_{write_j}$$

Using (15),

$$N_{rem-copy} = \frac{(n-1)(n+1)}{2n} \tag{18}$$

$N_{op-submit}$: *Average number of operations each transaction submits to remote sites.*

$$N_{op-submit} = N_r + N_w \tag{19}$$

$N_r (N_w)$ is the average number of read (write) operations submitted to remote sites by each transaction.

$$N_r = (1 - tr\_type\_prob) * N_{r,Q} + tr\_type\_prob * N_{r,U} \tag{20}$$

$N_{r,Q}$: Average number of remote read operations submitted by a query.
$N_{r,U}$: Average number of remote read operations submitted by an update transaction.

$$N_{r,Q} = (1 - P_L) * access\_mean$$

Using (5) for $P_L$,

$$N_{r,Q} = \frac{(n-1)}{2n} * access\_mean \tag{21}$$

$$N_{r,U} = (1 - data\_update\_prob) * (1 - P_L) * access\_mean$$

$$N_{r,U} = (1 - data\_update\_prob) * \frac{(n-1)}{2n} * access\_mean \tag{22}$$

Equations (21) and (22) can be combined into (20):

$$N_r =$$

$$\frac{(n-1)}{2n} * access\_mean * (1 - tr\_type\_prob * data\_update\_prob)$$

$$N_w =$$

$$tr\_type\_prob * data\_update\_prob * access\_mean * N_{rem-copy}$$

Remember that $N_{rem-copy}$ is the average number of remote copies of each data item accessed.

Using (18),

$$N_w = tr\_type\_prob * data\_update\_prob * access\_mean *$$

$$\frac{(n-1)(n+1)}{2n}$$

Equation (19), that is, the average number of operations each local transaction submits to the remote sites, becomes

$$N_{op-submit} =$$

$$\frac{(n-1)}{2n} * access\_mean * (1 + n * tr\_type\_prob * data\_update\_prob)$$