Distributed and Parallel Databases, 2, 405–436 (1994)

Processing Real-Time Transactions in a Replicated Database System

ÖZGÜR ULUSOY[1] oulusoy@bilkent.edu.tr
Department of Computer Engineering and Information Science, Bilkent University, Bilkent, Ankara 06533, Turkey

Received September 30, 1992; Revised September 10, 1993

Recommended by: A. Elmagarmid

Abstract. A database system supporting a real-time application has to provide real-time information to the executing transactions. Each real-time transaction is associated with a timing constraint, typically in the form of a deadline. It is difficult to satisfy all timing constraints due to the consistency requirements of the underlying database. In scheduling the transactions it is aimed to process as many transactions as possible within their deadlines. Replicated database systems possess desirable features for real-time applications, such as a high level of data availability, and potentially improved response time for queries. On the other hand, multiple copy updates lead to a considerable overhead due to the communication required among the data sites holding the copies. In this paper, we investigate the impact of storing multiple copies of data on satisfying the timing constraints of real-time transactions. A detailed performance model of a distributed database system is employed in evaluating the effects of various workload parameters and design alternatives on the system performance. The performance is expressed in terms of the fraction of satisfied transaction deadlines. A comparison of several real-time concurrency control protocols, which are based on different approaches in involving timing constraints of transactions in scheduling, is also provided in performance experiments.

Keywords: Real-time database systems, data replication, transaction scheduling, concurrency control, performance evaluation

1. Introduction

A *real-time database system* (RTDBS) is designed to provide real-time information to data-intensive applications. Each RTDB transaction is associated with a timing constraint, typically in the form of a deadline. It is difficult, in a RTDBS, to meet all timing constraints due to the consistency requirements of the underlying database. Concurrency control protocols proposed so far to preserve data consistency in database systems 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 primary consideration in scheduling RTDBS transactions is processing as many transactions as possible within their deadlines. A priority is assigned to each transaction based on its timing constraint to be used in ordering resource and data access requests of transactions. An extensive exploration of the issues in RTDBSs is provided in [32].

The transaction scheduling problem in RTDBSs has been addressed by a number of recent studies. The first attempt to evaluate the performance of scheduling algorithms in RTDBSs was provided in [1, 2]. Abbott and Garcia-Molina described and evaluated through simulation a group of real-time scheduling policies based on enforcing data consistency by using a two-phase locking concurrency control mechanism. An extended version of their work appeared recently in [4]. In [3], they provided a study of various algorithms for scheduling IO requests with deadlines. Carey et al. [11] and Chen et al. [13] also discussed some new approaches to priority-based IO scheduling. In [35] and [36], Sha et al. presented a new priority-based concurrency control protocol called *priority ceiling* (PC). The performance of this protocol PC was examined in [37] by using simulations. In [23], Huang et al. developed and evaluated several real-time policies for handling CPU scheduling, concurrency control, deadlock resolution, transaction wakeup, and transaction restart in RTDBSs. Later, their work was extended to the optimistic concurrency control method [24]. In [25], they proposed a new lock-based concurrency control protocol combining some existing schemes to capitalize on the advantages of each of those schemes. Haritsa et al. studied, by simulation, the relative performance of two well known classes of concurrency control algorithms (locking protocols and optimistic techniques) in a RTDBS environment [19, 22]. They presented and evaluated a new real-time optimistic concurrency control protocol through simulations in [20]. Son and Chang [40] investigated methods to apply the priority-ceiling protocol as a basis for real-time locking protocol in a distributed environment. Agrawal et al. [5] proposed a new locking approach, referred to as ordered sharing, which attempts to eliminate blocking of read and write operations in RTDBSs. In [42], Son et al. examined a priority-driven locking protocol which decomposes the problem of concurrency control into two subproblems, namely read-write synchronization and write-write synchronization, and integrates the solutions to two subproblems considering transaction priorities. Kim and Srivastava [26] proposed new multiversion concurrency control algorithms to increase concurrency in RTDBSs. Özsoyoğlu et al. [30] introduced new techniques to process database queries within fixed time quotas. Different degrees of accuracy of the responses to the queries can be achieved by using those techniques. In [43], we described several distributed, lock-based, real-time concurrency control protocols, and reported the relative performances of the protocols in a nonreplicated database environment.

Distributed databases fit more naturally in the decentralized structures of many RTDB applications that are inherently distributed (e.g., stock market, banking, command and control systems, and airline reservation systems). Distributed RTDBSs provide shared data access capabilities to transactions; i.e., a transaction is allowed to access data items stored at remote sites. While scheduling distributed RTDBS transactions, besides observing the timing constraints, it must also be provided 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 that in a single-site RTDBS.

In this paper, we focus our attention on the *data replication* aspect of distributed RTDBSs. In a *replicated database system* copies of data can be stored redundantly at multiple sites. The potential of data replication for high data availability and improved read performance is crucial to RTDBSs. On the other hand, data replication introduces its own problems. Access to a data item is no longer controlled exclusively by a single site, instead the access control is distributed across the sites each storing a copy of the data item. It is necessary to ensure that *mutual consistency* of the replicated data is provided; in other words, replicated copies must behave like a single copy. This is possible by preventing conflicting accesses on the different copies of the same data item, and by making sure that all data sites eventually receive all updates [18]. Multiple copy updates lead to a considerable overhead due to the communication required among the data sites holding the copies.

We investigated, in this study, the impact of storing multiple copies of data on satisfying the timing constraints of RTDBS transactions. A detailed performance model of a distributed RTDBS was employed in evaluating the effects of various workload parameters and design alternatives on the system performance. Several real-time concurrency control protocols were studied on a comparative basis. The locking-based protocols considered were the *priority-based conflict resolution* protocol (PB), which aborts a low priority transaction when one of its locks is requested by a higher priority transaction [1], the *priority inheritance* protocol (PI), which allows a low priority transaction to execute at the highest priority of all the higher priority transactions it blocks [35], and the *conditional priority inheritance* protocol (CP), which applies PB if a transaction holding a conflicting lock has not accessed many data items yet, otherwise it uses priority inheritance [25]. The *optimistic wait-50* protocol (OPT) performs a validation check for each committing transaction against the executing transactions. If half or more of the transactions conflicting with a committing transaction are of higher priority, the transaction is made to wait for the high priority transactions to complete; otherwise, it is allowed to commit while the conflicting transactions are aborted [20].

Although most of the previous works involving distributed database models assumed either *no-replication* [6, 28], or *full-replication* [15, 16, 31, 38, 39, 40, 41], some performance evaluation studies of *partially replicated* database systems were also provided [7, 12, 14, 29]. The impact of the level of data replication on the performance of conventional database systems was examined in those studies considering the average response time of the transactions and the system throughput to be the basic performance measures. It was found in those evaluations that increasing data replication usually leads to some performance degradation due to the overhead introduced by the replication. To the best of our knowledge, no performance evaluation work has appeared in the literature exploring data replication in RTDBSs.

Our performance model captures the basic characteristics of a distributed database system that processes transactions each associated with a timing constraint in the form of a deadline and a criticalness factor representing the importance of the transaction. A unique priority is assigned to each transaction based on its deadline and criticalness. The transaction scheduling decisions are basically affected by transaction priorities. The primary performance issue considered in our work is the satisfaction of transaction deadlines; more specifically, an answer to the following question is looked for: 'does replication of data always aid in satisfying real-time constraints of transactions?'. Various experiments were conducted to observe the performance characteristics of different applications as a function of the level of replication. Each application is distinguished by the type and data access distribution of the processed transactions. It was observed that replication is not attractive for update-oriented real-time applications due to the overhead of synchronizing updates on multiple copy data items. On the other hand, unless the majority of the transactions are update-oriented or the system load is high, it is preferable to store multiple copies (but not too many) of data. Finally, the effects of site failures were studied to estimate how much replication is needed to provide a reliable processing environment for real-time transactions of different applications.

In the next section, the distributed transaction structure and distributed execution model used in the simulations are presented. Section 3 describes our replicated database system model. The protocols used to control the concurrent transaction accesses to replicated data are described in Section 4. Section 5 provides the results of the performance evaluation experiments. The last section summarizes the conclusions of our work.

2. Distributed transaction execution model

Each distributed transaction exists in the system in the form of a master process that executes at the originating site of the transaction and a number of cohort processes that execute at various sites where the the copies of required data items reside. The transaction can have at most one cohort at each data site. The operations of a transaction are executed in a sequential manner, one at a time. For each operation executed, a global data dictionary is referred to find out the locations of the data item referenced by the operation. Each data site is assumed to have a copy of the global data dictionary. After determining which data sites should be accessed for the operation, a cohort process at each of those sites is initiated (if it does not exist already) by the master process to perform the operation in the name of the transaction. Previously created cohorts at those sites are just activated to perform the operation. After the successful completion of an operation, the next operation in sequence is executed by the appropriate cohort(s). When the last operation is completed, the transaction can be committed. Each transaction is assigned a globally unique priority based on its real-time constraints. This priority is carried by all of the cohorts of the transaction.

One-copy serializability in replicated database systems can be achieved by providing both concurrency control for the processed transactions and mutual consistency for the copies of a data item. In our replicated database system model, concurrency control is provided by any of the concurrency control protocols presented in the following sections, and mutual consistency of replicated data is achieved by using the *read-one, write-all-available* scheme [8]. The reason for selecting this replica control scheme is that alternatives like quorum-based approaches have the major drawback of turning read operations into multisite operations, even for local data [9, 12].[2] Based on the read-one, write-all-available approach, a read operation on a data item can be performed on any available copy of the data. On the other hand, in order to execute a write operation of a transaction on a data item, each transaction cohort executing at an operational data site storing a copy of the item is activated to perform the update on that copy (Figure 1).

The effects of a distributed transaction on the data must be made visible at all sites in an *all or nothing* fashion. The so called *atomic commitment* property can be provided by a commit protocol which coordinates the cohorts such that either all of them or none of them commit. In our model, atomic commitment of distributed transactions is provided by the centralized two-phase commit protocol [9].

```
operation_activation(op,T,D) {
  /* Operation op of transaction T will operate on data item D */
  if (∃ no operational site storing a copy of D)
    block until one of those sites becomes operational;
  if (op is a read)
    if (∃ a local copy of D)
      if (T has a local cohort C)
        submit op to C;
      otherwise {
        initiate a local cohort C;
        submit op to C;
      }
    otherwise
      if (∃ cohort of T at any operational site that holds a copy of D) {
        select a cohort C randomly among those cohorts;
        submit op to C;
      }
      otherwise {
        select a site randomly among operational remote sites storing a copy of D;
        initiate a cohort C of T on that site;
        submit op to C;
      }
  otherwise /* op is a write */
    for each operational site S storing a copy of D
        if (T has a cohort C at S)
          submit op to C;
        otherwise {
          initiate a cohort C of T at S;
          submit op to C;
        }
}
```

Figure 1. Operation activation procedure.

For the commitment of a transaction T, the master process of T is designated as *coordinator*, and each cohort process executing T's operations acts as a *participant* if its site is operational when the commit protocol is initiated. A periodical 'up-state' message broadcasted by each site is used in determining the current state (i.e., whether it is operational or failed) of that site. A site recovering from a failure executes an appropriate recovery procedure[3] to restore its database to a consistent and up-to-date state.

Following the execution of the last operation of transaction T, the coordinator (i.e., the master process of T) initiates Phase 1 of the commit protocol by sending a 'vote-request' message to all participants (i.e., cohorts of T) and waiting for a reply from each of them. If a participant is ready to commit, it votes for commitment, otherwise it votes for abort. An abort decision terminates the commit protocol for the participant. After collecting the votes of all participants, the coordinator initiates Phase 2 of the commit protocol. If all participants vote for commit, the coordinator broadcasts 'commit' message to them; otherwise, if any participant's decision is abort, it broadcasts an 'abort' message to the participants that voted for commit. The transaction is considered to have committed as soon as the coordinator broadcasts the 'commit' message to all participants. If a participant, waiting for a message from the coordinator, receives a 'commit' message, the execution of the cohort of T at that site finishes successfully. Following the successful commit of T, each cohort can write its updates (if any) into the local database of its site. An 'abort' message from the coordinator causes the cohort to be aborted. In that case the data updates performed by the cohort are simply ignored.

The blocking delay of two-phase commit (i.e., the delay experienced at both the coordinator site and each of the participant sites while waiting for messages from each other) is explicitly simulated in conducting the performance experiments.

3. A distributed RTDBS model

In the distributed system model, a number of data sites are interconnected by a local communication network. Each data site contains a transaction generator, a transaction manager, a resource manager, a message server, a scheduler, a buffer manager, and a recovery manager.

The transaction generator is responsible for generating the workload for each data site. The arrivals at a data site are assumed to be independent of the arrivals at the other sites. Each transaction in the system is distinguished by a globally unique transaction id. The id of a transaction is made up of two parts: a transaction number which is unique at the originating site of the transaction, and the id of the originating site which is unique in the system.

Each transaction is characterized by a *criticalness* and a *deadline*. The criticalness of a transaction is an indication of its level of importance [10]. It is assumed that each transaction is associated with one of m possible levels of criticalness (in this study, $m = 3$). The most critical transactions are assigned the highest level. Assignment of criticalness to a new transaction follows a uniform distribution; i.e., the criticalness of the transaction is chosen randomly from the set $\{1, 2, \ldots, m\}$. The deadline of a transaction specifies a certain time in the future the transaction has to be completed before. The deadline assignment method used in our RTDBS model is described later in this section. The transaction deadlines are *soft*; i.e., each transaction is executed to completion even if it misses its deadline. Criticalness and deadline are two independent characteristics of RTDB transactions [21, 23]. A close deadline does not necessarily imply more criticalness. The transaction manager at the originating site of a transaction T assigns a real-time priority to transaction T based on its criticalness (C_T), deadline (D_T), and arrival time (A_T).[4] The priority of transaction T is determined by the following formula:

$$P_T = \frac{C_T}{D_T - A_T}$$

The priority formula gives equal weight to criticalness and relative deadline.[5] If any two transactions originating from the same site carry the same priority, any scheduling decision between those transactions favors the more critical one; if the transactions are of the same criticalness as well, the transaction with closer deadline is scheduled first. To guarantee the global uniqueness of the priorities, the id of the originating site is appended to the priority of each transaction.

The transaction manager is responsible for creating a master process for each new transaction and specifying the appropriate sites for the execution of the cohort processes of the transaction. If there exist any local data in the access list of the transaction, one cohort will be executed locally. The coordination of the execution of remote cohorts is provided by the master process through communicating with the transaction manager of each cohort's site. To initiate the execution of each cohort the master process sends an 'initiate cohort' message to the relevant transaction manager. The initialization message contains the information required for the execution of the cohort (i.e., the id of the cohort's transaction and its priority). The transaction manager refers to this information to initiate the cohort. The transaction manager also provides the activation of each operation of a cohort executing at its site upon receiving an 'activate operation' message from the master process of the cohort.

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.

Access requests for data items are ordered by the scheduler on the basis of the concurrency control protocol executed. An access request of a cohort may result in blocking or abort of the cohort due to a data conflict with other cohorts executed concurrently. The scheduler at each site is 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. A *criticalness-based* FIFO page replacement strategy is used if no free memory space is available. The memory buffers allocated to transactions are organized into different lists and each list contains the buffers held by the transactions of the same criticalness. The buffer to replace is selected by FIFO rule from the buffer list of the lowest criticalness level among all nonempty lists.

Following the access, the data item is processed. When a cohort completes its data access and processing requirements, it waits for the master process to initiate two-phase commit. The master process commits a transaction only if all the cohort processes of the transaction run to completion successfully, otherwise it aborts and later restarts the transaction. A restarted transaction accesses the same data items as before, and is executed with its original priority. The cohorts of the transaction are reinitialized at relevant data sites.

IO and CPU services at each site are provided by the resource manager. IO service is required for reading or updating data items, while CPU service is necessary for processing data items and communication messages. Both CPU and IO queues are organized on the basis of real-time priorities, and preemptive-resume priority scheduling is used by the CPUs at each site. The CPU can be released by a cohort process either due to a preemption, or when the process commits or it is blocked/aborted due to a data conflict, or when it needs an IO or communication service. Communication messages are given higher priority at the CPU than data processing requests.

Table 1. Distributed RTDBS model parameters.

n	Number of data sites
local_db_size	database size originated at each site
N	number of copies of each data item
mem_size	main memory size 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 transaction interarrival time at a site
tr_type_prob	fraction of update type transactions
tr_length	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
mtbf	mean time between site failures
mttr	mean time to recover from a failure

The set of parameters described in Table 1 was used in specifying the configuration and workload of the distributed RTDBS.

Some of the concurrency control protocols to be discussed in Section 4 employ blocking in resolving data conflicts, thus, they are prone to blocking deadlocks. In those protocols, local deadlocks are detected by maintaining a local Wait-For Graph (WFG) at each site. WFGs contain the *wait-for* relationships among the transactions. Local deadlock detection is performed by the scheduler each time an edge is added to the graph (i.e., when a cohort is blocked). Assuming that a WFG is held in main memory, the processing cost of deadlock detection is considered to be proportional to the current number of edges constructing the WFG.[6]

Global deadlock is also a possibility in distributed systems. Two or more transactions can be in a deadlock chain waiting for each other to access the copies of the same data item or the copies of different data items stored at different sites. 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[7] of global deadlocks. The calculation of the processing cost of checking for a global deadlock is similar to that for local deadlocks; however, in this case, the size of the global WFG is taken into account. In addition to the processing cost of checking for a deadlock, the delay of communication messages carrying the local WFG information and the processing cost of those messages (at both the source and destination sites) are explicitly simulated by using parameters *comm_delay* and *mes_proc_time*, respectively. A deadlock is recovered from by selecting the lowest 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.

3.1. Data distribution model

We use a data distribution model which provides a partial replication of the distributed database. The model enables us to execute the system at precisely specified levels[8] of data replication. Each data item has exactly N copies in the distributed system, where $1 \leq N \leq n$. Each data site can have at most one copy of a data item. The remote copies of a data item are uniformly distributed over the remote data sites; in other words, the remote sites for the copies of a data item are chosen randomly. If the average database size at a site is specified by *db_size*,

$$db\_size = N * local\_db\_size$$

where *local_db_size* represents the database size originated at each site. Note that $N = 1$ and $N = n$ correspond to the no-replication and full-replication cases, respectively.

3.2. Deadline assignment

slack_rate is the parameter used in assigning deadlines to new transactions. The slack time of a transaction is chosen randomly from an exponential distribution with a mean of *slack_rate* times the estimated processing time of the transaction. While the transaction generator uses the estimation of transaction processing times in assigning deadlines, we assume that the system itself lacks the knowledge of processing time information. The deadline of a transaction T is determined by the following formula:

$$D_T = A_T + PE_T + S_T$$

where

$$S_T = expon(slack\_rate * PE_T)$$

A_T, PE_T, and S_T denote the arrival time, processing time estimate, and slack time of transaction T, respectively. The following formula provides the processing time estimate of T in an unloaded system.

$$PE_T = t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7$$

Each component of the formula is specified as follows.
t_1: Priority assignment delay.

$$t_1 = pri\_assign\_cost$$

t_2: Delay to locate the execution site(s) for the operations of T.

$$t_2 = tr\_length * lookup\_cost$$

lookup_cost corresponds to the processing cost of locating a single data item.

t_3: Delay due to cohort initialization messages.

$$t_3 = \mathit{nr\_coh\_sites}(T) * \mathit{mes\_proc\_time}$$

$\mathit{nr\_coh\_sites}(T)$ is the actual number of remote data sites on which T has cohorts to perform its operations. A message is sent to each remote site to initialize the cohort of the transaction at that site. Each message is processed before being sent, resulting in a total delay of $\mathit{nr\_coh\_sites}(T) * \mathit{mes\_proc\_time}$ units at its source.
t_4: Delay due to 'activate operation' and 'operation complete' messages for the remote operations.

$$t_4 = 2 * \mathit{rem\_op}(T) * (\mathit{mes\_proc\_time} + \mathit{comm\_delay} + \mathit{mes\_proc\_time})$$

$\mathit{rem\_op}(T)$ is the actual number of remote operations to be performed by T. Each 'activate operation' and 'operation complete' message has a communication overhead of $(\mathit{mes\_proc\_time} + \mathit{comm\_delay} + \mathit{mes\_proc\_time})$ time units.
t_5: Processing delay of the operations of T.

$$t_5 = \mathit{tr\_length} * \mathit{cpu\_time}$$

t_6: IO delay of the operations.
For a read-only transaction T,

$$t_6 = \begin{cases} \mathit{tr\_length} * \left(1 - \frac{\mathit{mem\_size}}{\mathit{db\_size}}\right) * \mathit{io\_time} & \text{if } \mathit{db\_size} > \mathit{mem\_size} \\ 0 & \text{otherwise} \end{cases}$$

$\mathit{db\_size}$ is the average size of database stored at each site. As specified above, $\mathit{db\_size} = N * \mathit{local\_db\_size}$
For an update transaction T,

$$t_6 = \begin{cases} \mathit{tr\_length} * \left(1 - \frac{\mathit{mem\_size}}{\mathit{db\_size}}\right) * \mathit{io\_time} + \mathit{w\_items}(T) * \mathit{io\_time} \\ \quad \text{if } \mathit{db\_size} > \mathit{mem\_size} \\ \mathit{w\_items}(T) * \mathit{io\_time} \\ \quad \text{otherwise} \end{cases}$$

$\mathit{w\_items}(T)$ refers to the actual number of data items updated by T.
t_7: Commit protocol overhead.

$$\begin{aligned} t_7 = {} & [\mathit{num\_coh\_sites}(T) * \mathit{mes\_proc\_time} + \mathit{comm\_delay} \\ & + 2 * \mathit{mes\_proc\_time} + \mathit{comm\_delay} \\ & + \mathit{num\_coh\_sites}(T) * \mathit{mes\_proc\_time}] \\ & + [\mathit{num\_coh\_sites}(T) * \mathit{mes\_proc\_time}] \end{aligned}$$

The terms contained within the first and the second square brackets correspond to overheads of Phase 1 and Phase 2 of the two-phase commit protocol, respectively. For Phase 1 of the protocol, $\mathit{num\_coh\_sites}(T) * \mathit{mes\_proc\_time}$ is the CPU time spent at the source of transaction to process the 'vote-request' messages before sending them to each of the remote cohorts; *comm_delay* is the communication delay of the messages before arriving at their destinations; $2 * mes\_proc\_time + comm\_delay$ is the delay due to processing the 'vote-request' message and processing the reply message before sending it and the communication delay of the replies sent to the master; and $num\_coh\_sites(T) * mes\_proc\_time$ is the time to read the reply messages from the remote cohorts. In determining the overhead of Phase 2, $num\_coh\_sites(T) * mes\_proc\_time$ is the processing time for the final decision messages before they are sent to remote cohorts.

3.3. Reliability issues

The distributed RTDBS model assumes that the data sites fail in accordance with an exponential distribution of inter-failure times. After a failure, a site stays halted during a repair period, again chosen from an exponential distribution. The means of the distributions are determined by the parameters *mtbf* (mean time between failures) and *mttr* (mean time to repair). The recovery manager at each site is responsible for handling site failures and maintaining the necessary information for that purpose. The communication network, on the other hand, is assumed to provide reliable message delivery and is free of partitions. It is 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 following sections details the reliability issues considered in our distributed system model.

3.3.1. Availability

Availability of a system specifies when transactions can be executed [17]. It is intimately related to the replica control strategy used by the system. For the read-one, write-all-available strategy, availability can be defined as the fraction of time (or probability) for which at least one copy of a data item is available to be accessed by an operation [29]. This strategy provides a high level of availability, since the system can continue to operate when all but one site have failed. In our simulations, a read or write operation on a data item D fails if no copy of D is available in the system. If N is the initial number of copies of D, a read/write operation on D succeeds if as many as $N-1$ of the copies are missing. If the last copy also vanishes, both read and write operations on D will fail. A transaction that issues an operation that fails is blocked until a copy of the requested data item becomes available.[9]

One method to measure the availability of an executing system is to keep track of the total number of attempted and failed operations experienced over a long period of time. It is possible to calculate the read and write operation availabilities separately as in [29], where the read (write) availability is defined and calculated as the total number of successful reads (writes) divided by the total number of read (write) requests. We prefer to use a more general calculation of system availability, which combines the read and write availabilities together in one formula. *Availability* in our model is defined by the following formula:

$$Availability = \frac{Total\ number\ of\ successful\ (read\ and\ write)\ operations}{Total\ number\ of\ (read\ and\ write)\ operation\ requests}$$

This formula is a convenient one to use in RTDBSs since both read and write availabilities are equally crucial to such systems, and thus they can be treated together.

3.3.2. *Site failure*

At a given time a site in our distributed system can be in any of three states: *operating*, *halted*, or *recovering*. A site is in the halted state if it has ceased to function due to a hardware or software failure. A site failure is modeled in a fail-stop manner; i.e., the site simply halts when it fails [33]. Following its repair, the site is transformed from the halted state to the recovering state and the recovery manager executes a predefined recovery procedure. A site that is operational or has been repaired is said to be in the operating state. Data items stored at a site are available only when the site is in the operating state.

A list of operating sites is maintained by the recovery manager at each site. The list is kept current by 'up-state' messages received from remote sites. An 'up-state' message is transmitted periodically by each operating site to all other sites. When the message has not been received from a site for a certain timeout period, the site is assumed to be down.

Our definition of data availability includes the case that an operation could fail after starting to execute. If a site processing a read operation of a transaction T fails before returning the result of the operation, the operation is submitted to another site, one which is in the operating state and storing a copy of the requested data item. If none of the operating sites has that item, the read operation fails and transaction T is blocked until a copy becomes available. A write operation of a transaction T is submitted to each operating site that stores a copy of the item to be updated. If any of those sites fails before completing the operation execution, the operation is just ignored at that site by the master process of T. If all the data sites involved in the execution of the operation fail, the operation is said to fail and transaction T is blocked.

3.3.3. *Site recovery strategy*

The recovery procedure at a site restores the database of that site to a consistent and up-to-date state. Our work does not simulate the details of site recovery; instead, it includes a simplified site recovery model which is sufficient for the purpose of estimating the impact of site failures on system performance.

The recovery manager at each site S maintains a log L_S for recovery purposes, which records the chronological sequence of write operations executed at S. Three types of records can exist in the log:

- <Start(T_i)> /* Transaction T_i has started at this site[10] */
- <T_i, D_j, *val*> /* Transaction T_i has updated data item D_j; the new value of D_j is *val* */

```
site_recovery($S_i$) {
  /* $S_i$ is the recovering site */
  Perform local recovery using $L_{S_i}$;
  Send a message to each site $S_j$ requesting the log $L_{S_j}$;
  Construct $DS$ by using the logs of the sites in operating state;
     /* $DS$ is the set of data items stored at $S_i$
        that have been updated since $S_i$ failed */
  for each data item $D \in DS$
     Update $D$ using the log of any site that stores a copy of $D$;
  Send an 'up-state' message to each remote site;
}
```

Figure 2. Site recovery procedure.

- <Commit(T_i)> /* Transaction T_i has committed */

Whenever a write operation is performed by a transaction, a log record for that write is created before the database is modified. At the commit time of a transaction, a commit record is written in the log at each participating data site. In the case of a transaction abort, the log records stored for that transaction are simply discarded. The recovery manager of a recovering site first performs local recovery by using the local log. Then, it obtains the logs kept at operating sites to check whether any of its replicated data items were updated while the site was in the halted state. It then refreshes the values of updated items using the current values of the copies stored at operational sites. This recovery procedure is summarized in Figure 2. Note that, if any data item stored at the recovering site has no other copies at operating sites, its consistency is provided through local recovery. We should state here that our recovery procedure is not able to eliminate completely the possibility of inconsistent execution due to site failures. Providing a very detailed model of failure which considers all possible cases that can lead to inconsistencies is beyond the scope of our work.

As discussed in [27], it is not necessary to write every log record to stable storage (disk) as soon as it is created. The transfer of log records from main memory to stable storage in blocks can safely be implemented. Each log record is written to the log tail (i.e., the last block of the log) stored in main memory. The log tail is written to the stable storage whenever it becomes full or right before the commit of a transaction (when the two-phase commit protocol starts to execute for the transaction).

4. Concurrency control protocols

The first three of the concurrency control protocols described below are based on two-phase locking. The management of locks for the data items stored at a site is provided by the scheduler of that 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 protocols are different in the way real-time priorities of transactions are involved in scheduling the lock requests.

An optimistic concurrency control protocol was also included in the set of evaluated protocols. In an optimistic protocol, the execution of each transaction consists of three phases: a read phase, a validation phase, and possibly a write phase. During the read phase, a transaction performs all its read and write operations without being blocked by any other transaction. The updates are performed on the local copies of data items and they are not accessible to other transactions. The validation phase checks whether the transaction execution can cause any inconsistency in the database. If a possible inconsistency is detected, the transaction is restarted. Otherwise, the transaction enters the write phase to reflect all the updates it performed into the database.

4.1. Priority-based conflict resolution protocol (PB)

This protocol resolves data conflicts always 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 priority of the cohort that is 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 cohort to abort and then restart the whole transaction.

If the lock on a data item is shared by a group of cohorts, a cohort C requesting an exclusive lock on the data item is blocked if any cohort sharing the lock has higher priority than the priority of C. Otherwise (if the priority of C is higher than the priorities of all lock sharing cohorts), the transactions of all the cohorts in the lock share group are aborted.

Assuming that no two transactions have the same priority, this protocol is deadlock-free since a high priority transaction is never blocked by a lower priority transaction.

4.2. Priority inheritance protocol (PI)

The priority inheritance method, proposed in [35], ensures 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.

Some other details related to the implementation of protocol PI in simulations are as follows. When a transaction, which has inherited a priority, is aborted due to a deadlock, it is restarted with its original priority. If the holder of a data lock is a group of cohorts sharing the lock, and if a high priority cohort C is blocked due to a conflict on that item, the cohorts which are in the shared lock group and have lower priority than C inherit the priority of C.

4.3. *Conditional priority inheritance protocol (CP)*

This protocol, proposed in [25], combines protocols PI and PB. When a cohort C is blocked by a lower priority cohort C', if the transaction of C' is near completion, it inherits the priority of C; otherwise, cohort C' (and thus its transaction) is aborted. The protocol assumes that the length of a transaction (i.e., the number of data items accessed by the transaction) is known in advance. The protocol has a threshold parameter h. At the time of a data conflict, if the remaining number of data items to be accessed by the transaction of the lock-holding cohort is less than or equal to threshold h, then protocol PI is applied; otherwise, protocol PB is used. The protocol is expected to reduce the blocking times with respect to PI, and to reduce the abort rate with respect to PB.

4.4. *Optimistic wait-50 protocol (OPT)*

An optimistic concurrency control protocol incorporating real-time priorities was proposed in [20]. The validation check for a committing transaction is performed against the executing transactions and if the write-set of the validating transaction intersects with the read-set of one of the executing transactions, the two transactions are said to be in conflict. This method of validation is called *broadcast commit.* The proposed protocol uses a *50 percent* rule as follows: If half or more of the transactions conflicting with a committing transaction are of higher priority, the transaction is made to wait for the high priority transactions to complete; otherwise, it is allowed to commit while the conflicting transactions are aborted. While the transaction is waiting, it is possible that it will be restarted due to the commit of one of the conflicting transactions with higher priority. The validation check for a transaction is performed at each data site where a cohort of the transaction has been executed.

5. Simulation experiments

The details of the replicated database system model and the transaction execution model described in previous sections were captured in a simulation program. The program was written in CSIM [34], which is a process-oriented simulation language based on the C programming language.

Simulation experiments were driven by the parameter values determined with the CPU/IO utilization formulas of the probabilistic model provided in [44]. The probabilistic model ensures that the parameter values are kept in appropriate ranges in obtaining a stable execution environment. Table 2 presents the default parameter values used in each of the experiments. All sites of the system were assumed identical and operating under the same parameter values. It was assumed that one CPU and one disk unit exist at each data site. Selection of the *cpu_time* and *io_time* parameter values aimed to obtain rather high and

Table 2. Distributed RTDBS model parameter values.

n	10
local_db_size	200
N	5
mem_size	500
cpu_time	8 msec (constant)
io_time	18 msec (constant)
comm_delay	5 msec (constant)
mes_proc_time	2 msec (constant)
pri_assign_cost	1 msec (constant)
lookup_cost	1 msec (constant)
iat	400 msec (exponential)
tr_type_prob	.5
tr_length	6 (constant)
data_update_prob	.5
slack_rate	5 (exponential)
mtbf	18,000 sec (exponential)
mttr	720 sec (exponential)

almost identical CPU and IO utilizations at each site. Neither a CPU-bound nor an IO-bound execution environment was intended to prevent the isolation of the effects of CPU contention or IO contention on the performance of the system. The small value of database size at each site[11] is to create a data contention environment which produces a high level of data conflicts among the concurrent transactions. This small database can be considered as the most frequently accessed fraction of a larger database.

Our expectation while choosing the values of the parameters *mtbf* and *mttr* was to obtain a system with high data availability. The simulation results of the availability versus data replication level experiment presented in [44] validated our expectations. For a nonreplicated system ($N = 1$), less than 5 percent of the operations failed due to site failures. With $N = 2$, the availability of data became more than 98 percent, and with $N = 4$, full availability was obtained.

The time period between consecutive 'up-state' messages transmitted by a data site was chosen as 100 seconds in our simulations. The log structure, used for recovery purposes, was assigned a blocksize of 50 records.

Replication of data was simulated explicitly by using the array *DataDictionary*, which specifies the mapping of data items to sites. Each index of the array corresponds to a single data item originating at any site. Considering the size of the database originating at each site (i.e., *local_db_size*), and the number of sites in the system (i.e., n), the size of the array is $n *$ *local_db_size*. Array entry *DataDictionary*$[i]$ contains the list of sites storing a copy of the i'th data item in the system ($1 \leq i \leq n *$ *local_db_size*). The array entries are filled at the beginning of each simulation by using the uniform data distribution assumption of Section 3.1. The data items to be accessed by each transaction are chosen randomly among the set of $n *$ *local_db_size* data items, and the data sites to execute the transaction operations are selected by referring to *DataDictionary* and using the operation execution procedure presented in Figure 1.

One possible performance metric that can be used in RTDB transaction scheduling is to determine the fraction of transactions that make their deadlines. Since our system processes transactions with different criticalness levels, we used a metric, *success-ratio*, that combines the performance measurements of all criticalness levels, in terms of the fraction of satisfied deadlines, using a specific weight for each level. This metric is defined as follows:

$$\textit{success-ratio} = \frac{\sum_{i=1}^{m} w_i * \textit{success-ratio}_i}{\sum_{i=1}^{m} w_i}$$

where
i: Criticalness level.
m: Total number of criticalness levels ($m = 3$ in our simulations).
w_i: Weight of criticalness level i.
success-ratio$_i$: Fraction of satisfied deadlines for the transactions of criticalness level i.

The determination of the weights of criticalness levels is highly dependent on the particular application environment [10]. We used linearly increasing weights; i.e.,

$$w_i = i, \qquad (i = 1, 2, \ldots, m)$$

For each experiment, the final results were evaluated as averages over 25 independent runs. Each run continued until 1000 transactions were executed at each data site. 90% 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. The following sections discuss only statistically significant performance results.

5.1. *Evaluating concurrency control protocols*

This experiment was conducted to evaluate the performance of the concurrency control protocols under different levels of transaction load. Mean time between successive transaction arrivals at a site (i.e., *iat*) was varied from 300 to 460 msec in steps of 40. This range of *iat* values corresponds to an expected CPU utilization of about .90 to .59 at each data site [40]. IO utilization is almost the same as CPU utilization with the parameter values chosen for the experiments. The performance results obtained with each protocol, in terms of *success-ratio*, are presented in Figure 3.

Our simulation program captures the effects of both data contention and resource contention. Data contention exists due to conflicting data access requests of transactions. Either transaction blocking or transaction restart is used by each concurrency control protocol to resolve a data conflict. Resource contention is due to the limited number of CPU/IO resources in the system. It results in queuing delay at each of those resources. Both data and resource contention at each data site are affected by the transaction load in the system. The number of data access conflicts among the concurrent transactions and the average length of CPU/IO queues increases as more transactions are processed at each site. Decreasing the level of transaction load (increasing *iat*) thus results in better performance for all concurrency control protocols tested in our performance experiments. As displayed in

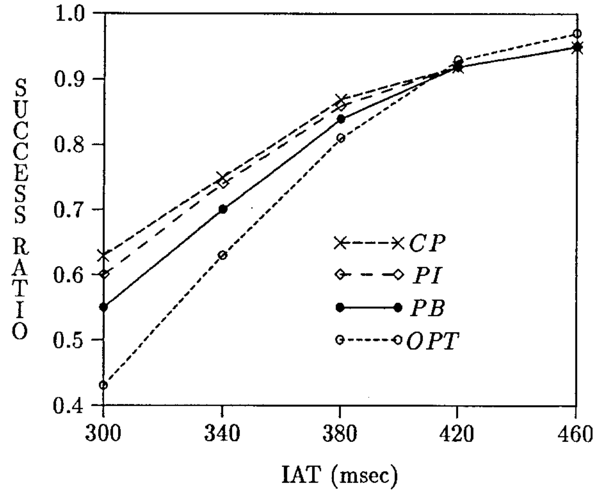

Figure 3. success-ratio vs *iat* (average transaction interarrival time (msec)).

Figure 3, between locking protocols PI and PB, the performance of priority inheritance protocol PI is somewhat better than that of priority-based conflict resolution protocol PB for a wide range of mean interarrival time. Remember that protocol PB 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 against protocol PB. Aborting a transaction which has already performed some write operations causes a considerable waste of IO/CPU resources at all the sites storing the copies of updated data. The results presented in Figure 3 for protocol CP, which combines protocols PI and PB, was obtained by setting threshold h of the protocol to 4. Figure 4 displays the performances of three locking protocols under different settings of threshold h. The performances of protocols PI and PB are independent of h. CP performs the same as PB when h is equal to 0, and the same as PI when h is set to 6 (i.e., the value of *tr_length*). The results presented in the figure were obtained with $iat = 300$ msec. Other possible settings of *iat* did not change the performance pattern of CP relative to PI and PB. The best performance with CP, under different levels of transaction load, was obtained for $3 \leq h \leq 5$. This result indicates that the strategy of protocol PB (i.e., aborting a low priority transaction if it is holding a conflicting lock) only works well if the transaction has processed not more than a few data items. It can be concluded that, in resolving a data conflict in a distributed RTDBS with replicated data, blocking the high priority transaction and executing the low priority one with the inherited high priority is preferable to aborting the low priority transaction unless the low priority transaction is in the early stages of its execution.

The optimistic wait-50 protocol OPT exhibits better performance than the locking protocols when the system is lightly loaded (i.e., for large *iat* values). No transaction is blocked due to data conflicts until commit time. Since the number of conflicts is small under low load levels, only a few transactions fail to be validated at commit time. On the other hand, when

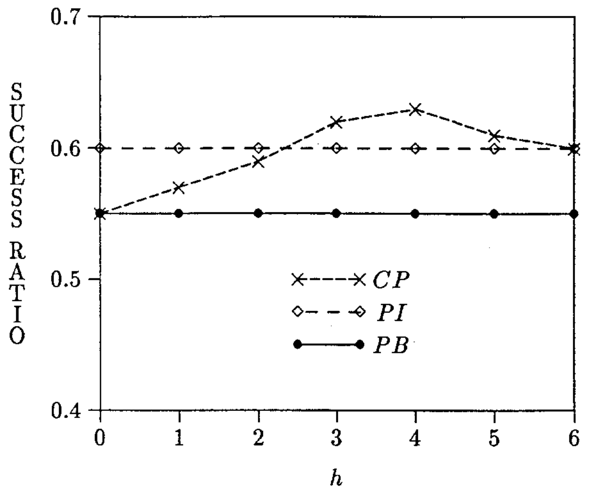

Figure 4. success-ratio vs threshold *h*.

the transaction load is high, the performance of protocol OPT becomes worse compared to the other protocols. As the number of data conflicts increases under heavier transaction load, the number of transaction restarts experienced with protocol OPT becomes more than that of the locking protocols.

Figure 5 presents the restart ratios (average number of restarts experienced by each transaction) under varying transaction loads for all four protocols. In protocol PI the only source of restarts is deadlock, while protocols PB, CP, and OPT may restart transactions to resolve data conflicts. Only a few more restarts are obtained with protocol CP compared to protocol PI, since CP applies priority inheritance in resolving most of the conflicts (as a result of setting the threshold h of CP to 4).

Haritsa et al. introduce a notion called *database access ratio* to be used in comparing the performances of concurrency control protocols [22]. The database access ratio is defined to be the maximum number of data items that could be simultaneously accessed by all the transactions in the system relative to the size of the database. This ratio was another parameter used in our experiments to vary data contention in the system in evaluating the concurrency control protocols. The number of distinct data items in our distributed database system is 2000 ($n * local\_db\_size$) and 6 data items are accessed by each transaction. It was shown in [44] that the total number of active transactions in the system does not exceed 50 even under the highest possible transaction load. Therefore, the highest database access ratio (with the database size of 2000 and the transaction length of 6) is $(50 * 6)/2000 = 0.15$. We evaluated the concurrency control protocols for different values of the database access ratio by varying the value of parameter *tr_length* (i.e., the number of data items accessed by each transaction). The mean interarrival time value (*iat*) was fixed at 400 msec and the same value was assumed for the maximum transaction population (i.e., 50) with each *tr_length* value considered. The range of *tr_length* values employed was [2,10], which corresponds

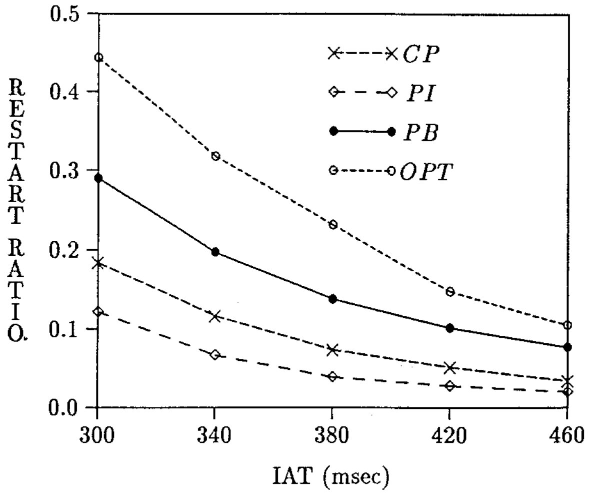

Figure 5. Average number of restarts per transaction vs *iat*.

to a database access ratio of 0.05 to 0.25. The results are displayed in Figure 6 for protocols PI and OPT. The reaction of protocols PB and CP to the change in the database access ratio was similar to that of protocol PI, thus, PI was selected as representative for the locking protocols. As the database access ratio gets higher, both PI and OPT perform worse. At low values of database access ratio (i.e, at low contention levels) OPT is observed to perform a little bit better than PI; however, PI outperforms OPT at higher database access ratios. This result is in agreement with our previous results provided above that were obtained by using another parameter (i.e., *iat*) in varying the level of data contention. On the other hand, it was shown by Haritsa et al. that optimistic protocols are superior to locking protocols at high database access ratios [19, 22]. This result is different from what we obtained in our experiments. However, the experiments of Haritsa et al. were performed in a RTDBS that discards late transactions (i.e., the deadlines are firm) and most of their simulation results were obtained under the assumption that the system has infinite resources. These assumptions, most probably, are the source of the difference between their results and ours; because, when they processed the transactions in a finite resource system, with soft deadlines [19] (as in our model), they found that the locking protocol performs better than the optimistic one, which confirms our findings.

The results provided so far were obtained by employing the one-at-a-time (sequential) transaction execution model detailed in Section 2. Another execution model in which the cohorts of a transaction act in parallel is discussed in [43]. In this model the master process of a transaction spawns cohorts all together, and the cohorts are executed in parallel. The master process sends to each remote site a message containing an (implicit) request to spawn a cohort, and the list of all operations of the transaction to be executed at that site. The assumption here is that the operations performed by one cohort are independent of the results of the operations performed at the other sites. The sibling cohorts do not have to transfer

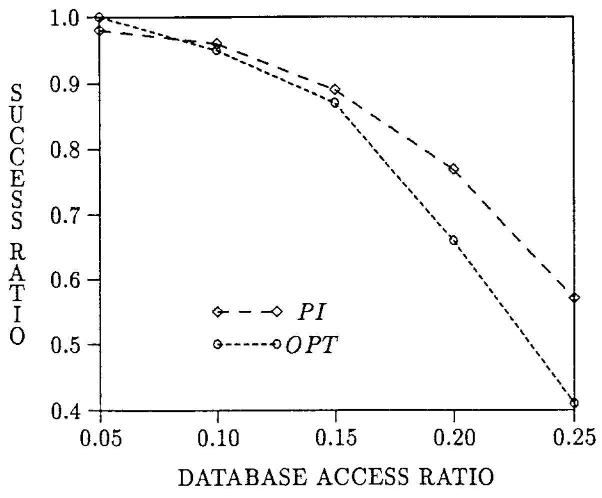

Figure 6. success-ratio vs database access ratio.

information to each other. A cohort is said to be completed at a site when it has performed all its operations. A completed cohort informs the master process by sending a 'cohort complete' message. The master process can start the two-phase commit protocol when it has received 'cohort complete' messages from all the cohorts. Various experiments were performed with the parallel execution model. It was observed that the real-time performance is much better compared to the one-at-a-time model due to less communication delay and shorter transaction life. However, the comparative performances of the concurrency control protocols were not affected.

5.2. Impact of level of data replication

In this section, we evaluate how successful the transactions are in satisfying their deadlines under different levels of data replication. We consider four different application environments in conducting data replication experiments. As summarized in Table 3 each application environment is characterized by the fraction of update transactions processed, and the distribution of accessed data items. The majority of transactions in the first two applications are read-only (RO), while the last two applications are dominated by update (UP) transactions. In the first and third applications most of the data items accessed by transactions originate locally (LOC); on the other hand, for the other applications the originating sites of accessed data items are chosen from a uniform distribution, thus, accesses to data items originating at remote (REM) sites dominate, since there exist more than two sites in the system. Remember that in the experiments of Section 5.1, 50 percent of the transactions were update type (as specified in Table 2) and data accesses of each transaction were uniformly distributed over all sites.

Table 3. Application environments considered in data replication experiments.

Application Type	Update Transaction Percentage	Data Access Distribution
RO_LOC	25%	75% local origin 25% remote origin
RO_REM	25%	uniform over all database
UP_LOC	75%	75% local origin 25% remote origin
UP_REM	75%	uniform over all database

In evaluating the effects of level of data replication on system performance, the number of replicas of each data item (N) was varied from 1 to n ($n = 10$). The mean interarrival time value (*iat*) was fixed at 400 msec. Figure 7 presents the results obtained xent application environments with concurrency control protocol PI.

With the first two application types, which represent an execution environment where the majority of transactions are queries, the fraction of satisfied deadlines is at a high level compared to the other application types. The number of conflicts among the transactions increases when the fraction of update operations becomes higher, which results in a degradation in the performance of the RTDBS.

With application type *RO_REM*, an improvement in the performance is possible up to a certain point (7 replicas in this example) by increasing the data replication level. This improvement can be explained by the increasing number of local read operations eliminating the cost of inter-site communication. For more replicas, further improvement is not possible since the performance advantage gained by the local read operations is outweighed by the overhead of multiple copy updates. For application type *RO_LOC*, on the other hand, since most of the transactions access locally originated data items, the increase in the number of local read operations by providing more data replicas is not enough to affect the performance. The *success-ratio* graph for $N \geq 2$ is almost flat. The performance level achieved for no-replication case ($N = 1$) is not as high as that obtained with other values of N. The worse performance obtained by maintaining a single copy of each data item can be explained by the unavailability of data during down periods as a result of site failures. It was shown in [44] that having a couple of data copies is effective in preventing the effects of site failures on system performance for all application types.

Due to the local data accesses the *success-ratio* obtained with *RO_LOC* is better than that with *RO_REM*, except under high levels of data replication where the same execution conditions exist for both application types.

For application types *UP_LOC* and *UP_REM*, where the majority of transactions are of update type, a considerable degradation in performance is observed if the level of data replication is increased beyond 3. The overhead of update synchronization among the multiple copies of updated data increases with each additional data copy. The difference between the performance results of those two application types is due to accessing more local data items with *UP_LOC*. At full replication ($N = 10$), the same performance is

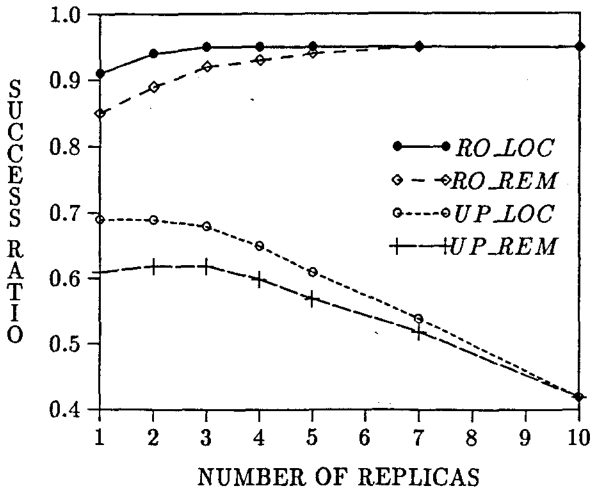

Figure 7. success-ratio vs *N* (number of data replicas) with different application types.

exhibited with both application types, since all read operations are performed on local data copies.

We conclude that data replication can reduce the effects of site failures and provide faster response to real-time queries; however, the primary factor determining the performance is the overhead of update synchronization among data replicas. Except for the query-dominant application environment where the transactions usually require remote access (i.e., application type *RO_REM*), the best results[12] in general were obtained when each data item had 2 or 3 copies in the system. For application type *RO_REM* it is possible to improve the performance by increasing the number of copies beyond 2 or 3.

When the experiment was repeated with the other concurrency control protocols, it was observed that although the protocols generate somewhat different *success-ratio* results under the same conditions, qualitatively the results are in agreement with those above. However, the relative performances of the protocols show some differences under different levels of data replication. With application type *RO_LOC*, the performance results obtained by protocols CP, PI, PB, and OPT were not distinguishable from each other. All the protocols perform equally well under different levels of data replication when the system is dominated by queries accessing only local data. Figure 8 presents the results obtained with protocols CP, PI, PB, and OPT under application type *RO_REM*. All protocols behave similarly as the level of replication changes; increasing the number of replicas results in better performance up to a certain replication level. Comparing the results obtained for each protocol, one can see that under low levels of replication, resolving data conflicts by using transaction restart leads to better performance than employing transaction block. Protocol OPT exhibits the best performance if the number of copies of each data item is not many. Since the system is dominated by read-only transactions, the number of data conflicts is small; and, as discussed in the preceding section, protocol OPT performs well when the level of data conflicts is

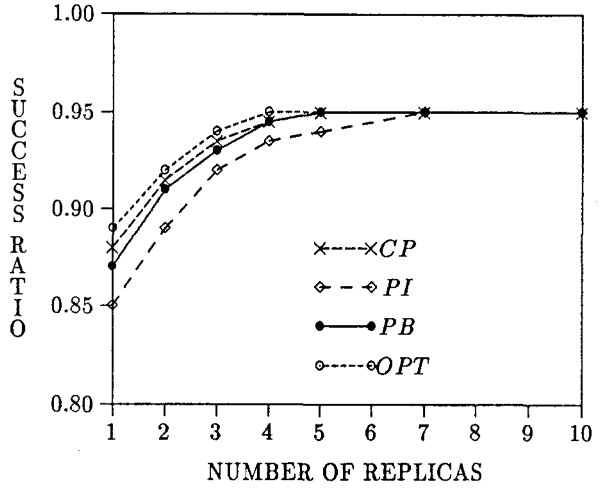

Figure 8. success-ratio vs N (number of data replicas) with application type *RO_REM*.

low. The other restart-based protocol PB also provides better performance than protocol PI unless data replication is high. The best performance with CP is obtained when threshold h is assigned a very small value (i.e., when each lock conflict is resolved by applying PB unless the low priority lock holding transaction is near completion). As shown in the figure, where the results for CP were obtained by setting h to 2, CP provides a little bit improvement in the performance of PB under low levels of replication. As the level of replication increases, the performance of the protocols becomes closer, and near full replication all the protocols behave similarly. The improvement in the performance of PI at each extra data copy is greater than that of the other protocols. This result is due to the fact that aborting a transaction becomes more expensive (with protocols PB, CP, and OPT) as the number of copies of the data items updated by the transaction increases.

The comparative performance results of the protocols obtained for different data replication levels with application types *UP_LOC* and *UP_REM* are completely different from those of *RO_REM* discussed above. The *success-ratio* results for the concurrency control protocols with *UP_REM* are displayed in Figure 9. Protocol PI, in this case, provides better performance than protocols PB and OPT for the entire N range explored. Protocol CP, with a threshold h value of 4, can provide a slight improvement in the performance of PI. It can be seen from the figure that for an application where the majority of transactions are update type, having multiple copies of data items does not help transactions satisfy their timing constraints. Increasing the level of data replication results in worse performance for all the concurrency control protocols employed. However, the blocking-based protocol PI seems to be the one that is affected least by that increase. It can be concluded that the overhead of executing a blocking-based concurrency control protocol is less than that of a restart-based one when update transactions dominate in the system. This result is similar to what we observed in the experiment of Section 5.1 (where almost half of the transactions were update

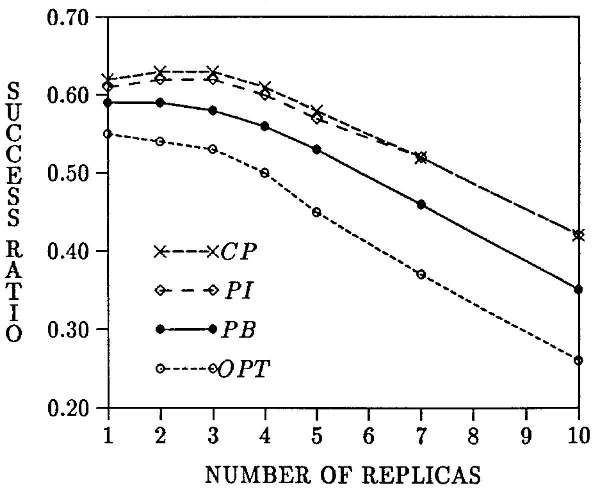

Figure 9. success-ratio vs N (number of data replicas) with application type *UP_REM.*

type) under high levels of transaction load. As the level of replication increases, the performance difference becomes more between the protocol that uses blocking as the conflict resolution strategy (i.e., PI) and the protocols that employ transaction restart in resolving data conflicts (i.e., PB and OPT). Similar performance characteristics were obtained with application type *UP_LOC.*

A quorum-based replica control scheme was also employed in our simulations. In this scheme, a read request on a data item is honored only when a read quorum of q_r copies can be accessed. Similarly, to perform a write operation on a data item, a write quorum of q_w copies must be updated. The conditions $2q_w > N$ and $q_r + q_w > N$ ensure that each write quorum of a data item has at least one copy in common with every read quorum and every write quorum of the item.[13] A version number is maintained with each copy, which is initially 0. Performing a write operation of a transaction requires each cohort of the transaction executing on a copy of a write quorum to send the version number of that copy to the master process. After collecting the version numbers of all copies, the master process determines the maximum version number, increments it, and broadcasts that version number to the relative sites to be assigned to each copy. A cohort performing a read operation returns the version number of the data item copy along with the data value. The master process selects the copy in the read quorum with the highest version number which gives the most recent value of the data item. A read (write) operation on a data item fails if q_r (q_w) copies are not available when requested. The availability can be defined as the fraction of time (or probability) for which we are able to form a quorum.

The experiment that evaluates the effects of level of data replication on system performance was repeated by employing the quorum-based replica control scheme. Comparing the results illustrated in Figure 10 to those obtained with the read-one, write-all-available scheme (Figure 7), it can be seen that for query-dominant application environments (i.e.,

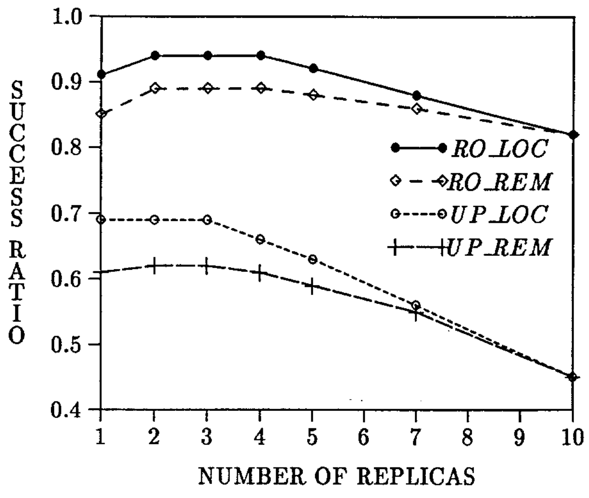

Figure 10. success-ratio vs *N* with a quorum-based replica control scheme.

RO_LOC and *RO_REM*) the quorum-based scheme leads to noticeably worse performance under high levels of replication. Even if there exists a local copy of the data item requested for a read operation, some remote copies of the item must also be accessed to build a read quorum. Thus, more communication overhead is incorporated in the quorum-based scheme. Under update-oriented transaction execution environments (i.e., *UP_LOC* and *UP_REM*), although the quorum-based scheme was expected to perform much better than the read-one, write-all-available scheme (because less number of copies is involved in each write operation), there is very little additional gain in performance. One fact that can lead to this result is the overhead experienced with the quorum-based scheme due to the communication messages carrying the version numbers of data item copies that need to be adjusted at each write operation. The relative performance of the concurrency control protocols was not sensitive to the replica control scheme employed. The results discussed lead to the general observation that the basic consideration in the selection of the replica control scheme to implement should be the workload characteristics (i.e., read-write ratios) of the underlying application.

Before closing this section, we want to make a point about the comparison of our results to those obtained in some previous works of evaluating the effects of data replication on the performance of conventional (non real-time) database systems. In terms of system throughput and/or average response time of transactions, it was agreed in general that increasing replication leads to some performance degradation due to the overhead of updating data copies [7, 12, 14]. Taking the advantage of replication is possible only under certain conditions, like light transaction loads or few number of data updates. As stated before, our evaluations in RTDBSs considered the fraction of satisfied transaction deadlines to be the basic performance measure. It was observed in our work that data replication does not always lead to poor performance in processing real-time transactions. Our results showed

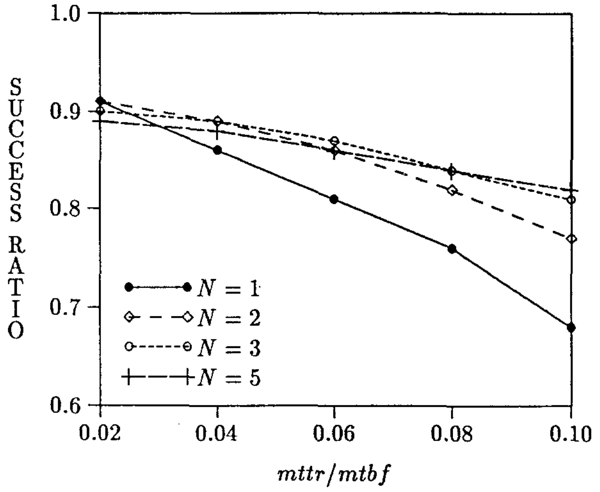

Figure 11. *success-ratio* vs *mttr/mtbf.*

that, unless the majority of the application transactions are update-oriented and/or most data accesses are remote, having a few copies of data is preferable to not replicating it at all. Another aspect of replication that has to be considered is the reliability provided in the face of failures. Even if the performance gain obtained by replicating data might be small, a reliable system is very crucial to RTDBSs. The issue of reliability is addressed in the next section.

5.3. Performance under different reliability levels

Another interesting experiment was the evaluation of the performance under different values of the failure parameter *mtbf*; in other words, under different system reliability levels. The larger the value of *mtbf*, the more reliable is the system. This experiment was repeated for different levels of data replication. Figure 11 illustrates the performance results for N values of 1, 2, 3, and 5. PI is the concurrency control protocol employed in this experiment. The range of values used for parameter *mtbf* was [7200 sec, 36000 sec], which corresponds to a *mttr/mtbf* ratio of [.10,.02]. The *tr_type_prob* value chosen for this experiment was 0.5, and data accesses of each transaction were uniformly distributed over all sites. The value of parameter *iat* was again fixed at 400 msec. It can easily be seen from the figure that the effect of failure frequency on the performance of the system increases as the level of data replication decreases. If the distributed database system is nonreplicated ($N = 1$), *success-ratio* sharply decreases as the sites fail more often. Availability of data items is increased by having multiple copies of the items at different sites. The performance, in terms of *success-ratio*, is less affected by failures with each additional copy of the data items; however, some degradation is still seen. The graph for $N \geq 4$ is relatively flat, because operation failures due to data unavailability were not observed for high levels of replication even under the most frequent site failure case tested. The results for $N = 5$ were chosen as representative for this situation. The slight degradation in performance as *mttr/mtbf* increases can be explained by the overhead of more frequent site recoveries, and the fact that there is less useful work at each site due to increasing number of down periods. It should also be noted that, as site failures become more frequent, it becomes more desirable to have high levels of data replication to obtain better performance. If a distributed RTDBS experiences site failures quite often, having a few copies of data items at multiple sites helps a lot in improving the performance.

6. Conclusions

The primary performance consideration in a real-time database system (RTDBS) (i.e., a database system that processes transactions with timing constraints) is to provide schedules that maximize the number of satisfied timing constraints. Data replication is an important concept that needs to be explored in studying the performance aspects of distributed RT-DBSs. It may be desirable to replicate data because of certain advantages provided, such as high data availability and potentially improved read performance. However, under certain conditions, the overhead of synchronizing multiple copy updates can become a considerable factor in determining performance. In this paper, we tried to identify the conditions under which data replication can help real-time transactions satisfy their timing constraints.

A detailed model of a distributed RTDBS was employed in evaluating the impact of data replication on the system performance. Each transaction processed in the system was associated with a timing constraint in the form of a deadline and a criticalness factor representing the importance of the transaction. A unique priority was assigned to each transaction based on its criticalness and deadline. The performance of the system was specified in terms of the fraction of satisfied transaction deadlines. The criticalness of satisfied deadlines was also considered in determining performance. Four priority-based concurrency control protocols, three locking-based and one optimistic, were employed in performance evaluations. The protocols are different in the way real-time priorities of transactions are involved in scheduling data access requests. The priority-based conflict resolution protocol (PB) aborts a low priority transaction when one of its locks is requested by a higher priority transaction. The priority inheritance protocol (PI) always blocks a lock-requesting transaction and allows a low priority transaction to execute at the highest priority of all the higher priority transactions it blocks. The conditional priority inheritance protocol (CP) resolves a lock conflict by applying either one of protocols PB and PI, depending on the age of the lock-holding transaction. The wait-50 optimistic protocol (OPT) performs a conflict check at the commit time of a transaction, and in the case of a conflict if at least 50 percent of the conflicting transactions have higher priority than the committing transaction, the transaction is blocked until the high priority transactions complete; otherwise, it is allowed to commit while the conflicting transactions are aborted. Protocol PI employs only transaction blocking in resolving data conflicts, while the other protocols use both transaction blocking and transaction restart.

Different application types were considered in evaluating the effects of level of data replication on satisfying transaction deadlines. Each type considered is characterized by the fraction of update transactions processed and the distribution of accessed data items (local vs remote). In execution environments where queries predominate and the data items at all sites are accessed uniformly, increasing the level of replication helped transactions meet their deadlines. For the application types where the majority of processed transactions are of update type, having many data copies was not attractive. This result was due to the fact that the overhead of update synchronization among the multiple copies of updated data increases with each additional data copy. Concurrency control protocols PB and OPT, which employ restarts in scheduling, exhibited better performance than protocol PI in query-based application environments when the level of data replication was low. Under the same conditions, the execution of protocol CP proved that it is possible to improve the performance of protocol PB if protocol PI is applied once the lock-holding transaction in a conflict is near completion.

With update-oriented applications protocol PI outperformed protocols PB and OPT, leading to the result that the overhead of executing a blocking-based concurrency control protocol is less than that of a restart-based one when the update transactions dominate in the system. Protocol PI becomes more preferable as the level of data replication increases, since the performance of restart-based protocols is affected more negatively by the increased overhead of multiple copy updates. Aborting a transaction becomes more expensive as the number of copies of the data items updated by the transaction increases. By employing protocol CP, it was shown that aborting a lock-holding transaction should be considered only in the case that the transaction is at the early stages of its execution.

[44] provides an evaluation of some concurrency control protocols in a single-site RTDBS. In that work, protocol PB was shown to perform better than protocol PI under various conditions. However, as discussed above, in a distributed RTDBS PB can beat PI only under query-based application environments and when the level of data replication is low. These two observations lead to the conclusion that restart-based protocols (like PB) are superior to blocking-based protocols (like PI) as long as the overhead of transaction aborts is not high. As the data becomes more distributed and replicated, the increased overhead of transaction aborts causes PB to perform worse than PI. Huang et al. also found that PB performs better than PI in a single-site RTDBS [25]. Similar to our findings in this paper, they showed that protocol CP can improve the performance of PB by applying PI for the transactions near completion. The performance of protocol OPT in a single-site RTDBS was found to be good only under light transaction loads [44]. This result agrees with our findings for the performance of OPT in a replicated RTDBS.

We also studied the impact of site failures on system performance under different system reliability levels. Investigating the effectiveness of data replication in preventing the effects of site failures, we observed that replication turns out to be more desirable as site failures become more frequent.

The results of our performance experiments led to the following general observation: the optimum number of data replicas to provide the best performance in RTDBSs depends upon the fraction of update operations required by the application, the distribution of accessed data items, and the reliability of data sites. In general, as few as 2 or 3 copies appeared to be a good choice under the parameter ranges explored.

Acknowledgment

I would like to thank Prof. Geneva G. Belford and Prof. Benjamin Wah of University of Illinois, and the anonymous reviewers for their comments and suggestions on previous versions of the paper.

Notes

1. This work was initiated while the author was at the Computer Science Department, University of Illinois at Urbana-Champaign.
2. The results of the experiments performed with a quorum-based scheme are provided in Section 5.2.
3. The recovery procedure is detailed in Section 3.3.3.
4. A dynamic priority assignment policy, which evaluates the transaction priorities continuously, was not implemented due to the considerable overhead incurred by calculation of the priorities whenever needed.
5. $D_T - A_T$ specifies the relative deadline of transaction T.
6. In simulations, the value of proportionality factor was taken as 0.1 msec [44]; i.e., (0.1 msec $*$ the number of edges in the WFG) is the CPU time spent by the scheduler checking for a deadlock.
7. The time period between consecutive global deadlock detection was chosen as 10 seconds in the simulations.
8. The level of replication corresponds to the number of copies that exist for each data item.
9. Aborting the failed transaction was another method considered in our simulations. It is provided by this method that the transactions blocked by a failed transaction do not have to stay blocked until the data item required by that transaction becomes available. However, the performance of the system was not affected considerable by the change in the strategy handling failed transactions, because, as discussed in the Simulation Experiments section, the number of failed operations due to unavailability of data was observed to be very few in the experiments.
10. At the originating site of T_i, this record is placed in the local log upon the arrival of T_i. At a remote site, the record is inserted in the log when a cohort of T_i is submitted to that site.
11. Average database size at each site is *db_size* $= N *$ *local_db_size* $=$ 1000 data items.
12. For the parameter ranges explored.
13. The values of q_r and q_w in our experiments were determined by using the following two equations:

$$q_w = \left\lceil \frac{N+1}{2} \right\rceil, \quad q_r + q_w = N + 1$$

References

1. Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation," *14th International Conference on Very Large Data Bases*, 1988, pp. 1–12.
2. Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data," *15th International Conference on Very Large Data Bases*, 1989, pp. 385–396.
3. Abbott, R. and Garcia-Molina, H., "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *11th Real-Time Systems Symposium*, 1990, pp. 113–124.

4. Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Transactions on Database Systems*, vol. 17, pp. 513–560, 1992.
5. Agrawal, D., El Abbadi, A. and Jeffers, R., "Using Delayed Commitment in Locking Protocols for Real-Time Databases," *ACM SIGMOD Conference*, 1992, pp. 104–113.
6. Balter, R., Berard, P., and Decitre, P., "Why Control of Concurrency Level in Distributed Systems is More Fundamental Than Deadlock Management," *1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1982, pp. 183–193.
7. Barbara, D. and Garcia-Molina, H., "How Expensive is Data Replication? An Example," *2nd International Conference on Distributed Computing Systems*, 1982, pp. 263–268.
8. Bernstein, P.A. and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, vol. 9, pp. 596–615, 1984.
9. Bernstein, P.A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
10. Biyabani, S.R., Stankovic, J.A., and Ramamritham, K., "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *9th Real-Time Systems Symposium*, 1988, pp. 152–160.
11. Carey, M.J., Jauhari, R., and Livny, M., "Priority in DBMS Resource Scheduling," *15th International Conference on Very Large Data Bases*, 1989, pp. 397–410.
12. Carey, M.J. and Livny, M., "Conflict Detection Tradeoffs for Replicated Data," *ACM Transactions on Database Systems*, vol. 16, pp. 703–746, 1991.
13. Chen, S., Stankovic, J.A., Kurose, J., and Townley, D., "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *Real-Time Systems Journal*, vol. 3, pp. 307–336, 1991.
14. Ciciani, B., Dias, D.M., and Yu, P.S., "Analysis of Replication in Distributed Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 247–261, 1990.
15. Galler, B.I. and Bos, L., "A Model of Transaction Blocking in Databases," *Performance Evaluation*, vol. 3, pp. 95–122, 1983.
16. Garcia-Molina, H., "Reliability Issues for Fully Replicated Distributed Databases," *IEEE Computer*, vol. 15, pp. 34–42, 1982.
17. Garcia-Molina, H., "The Future of Data Replication," *5th Symposium on Reliable Distributed Systems*, 1986, pp. 13–19.
18. Garcia-Molina, H. and Abbott, R.K., "Reliable Distributed Database Management," *Proceedings of the IEEE*, vol. 75, pp. 601–620, 1987.
19. Haritsa, J.R., Carey, M.J., and Livny, M., "On Being Optimistic About Real-Time Constraints," *ACM SIGACT-SIGMOD-SIGART*, 1990, pp. 331–343.
20. Haritsa, J.R., Carey, M.J., and Livny, M., "Dynamic Real-Time Optimistic Concurrency Control," *11th Real-Time Systems Symposium*, 1990, pp. 94–103.
21. Haritsa, J.R., Carey, M.J., and Livny, M., "Value-Based Scheduling in Real-Time Database Systems," Technical Report No. 1024, Dept. of Computer Science, University of Wisconsin-Madison, 1991.
22. Haritsa, J.R., Carey, M.J., and Livny, M., "Data Access Scheduling in Firm Real-Time Database Systems," *Real-Time Systems*, vol. 4, pp. 203–241, 1992.
23. Huang, J., Stankovic, J.A., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *10th Real-Time Systems Symposium*, 1989, pp. 144–153.
24. Huang, J., Stankovic, J.A., Ramamritham, K., and Towsley, D., "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *17th International Conference on Very Large Data Bases*, 1991, pp. 35–46.
25. Huang, J., Stankovic, J.A., Ramamritham, K., and Towsley, D., "On Using Priority Inheritance In Real-Time Databases," *12th Real-Time Systems Symposium*, 1991, pp. 210–221.
26. Kim, W. and Srivastava, J., "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling," *12th Real-Time Systems Symposium*, 1991, pp. 222–231.
27. Korth, H.F. and Silberschatz, A., *Database Systems Concepts*, 2nd Edition, McGraw-Hill, 1991.
28. Lin, W. and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking," *9th International Conference on Very Large Data Bases*, 1983, pp. 109–119.
29. Noe, J.D. and Andreassian, A., "Effectiveness of Replication in Distributed Computer Networks," *7th International Conference on Distributed Computing Systems*, 1987, pp. 508–513.
30. Özsoyoğlu, G., Özsoyoğlu, Z.M., and Hou, W.C., "Research in Time and Error-Constrained Database Query Processing," *7th IEEE Workshop on Real-Time Operating Systems and Software*, 1990, pp. 32–38.

31. Özsu, M.T., "Performance Comparison of Distributed vs Centralized Locking Algorithms in Distributed Database Systems," *5th International Conference on Distributed Computing Systems*, 1985, pp. 254–261.
32. Ramamritham, K., "Real-Time Databases," to appear in *International Journal of Distributed and Parallel Databases*, 1993.
33. Schlicting, R.D. and Schneider, F.B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, vol. 1, pp. 222–238, 1983.
34. Schwetman, H., "CSIM: A C-Based, Process-Oriented Simulation Language," *Winter Simulation Conference*, 1986, pp. 387–396.
35. Sha, L., Rajkumar, R., and Lehoczky, J., "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, vol. 17, no. 1, pp. 82–98, 1988.
36. Sha, L., Rajkumar, R., and Lehoczky, J., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, pp. 1175–1185, 1990.
37. Sha, L., Rajkumar, R., Son, S.H., and Chang, C.H., "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, pp. 793–800, 1991.
38. Singhal, M., "A Fully-Distributed Approach to Concurrency Control in Replicated Database Systems," *12th International Computer Software and Applications Conference*, 1988, pp. 353–360.
39. Singhal, M., "Update Transport: A New Technique for Update Synchronization in Replicated Database Systems," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1325–1336, 1990.
40. Son, S.H. and Chang, C.H., "Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, 1990, pp. 124–131.
41. Son, S.H. and Kouloumbis, S.,"Performance Evaluation of Replication Control Algorithms for Distributed Database Systems," Technical Report, CS-TR-9-11, University of Virginia, 1991.
42. Son, S.H., Park, S., and Lin, Y., "An Integrated Real-Time Locking Protocol," *8th International Conference on Data Engineering*, 1992, pp. 527–534.
43. Ulusoy, Ö. and Belford, G.G., "Real-Time Lock Based Concurrency Control in a Distributed Database System," *12th International Conference on Distributed Computing Systems*, 1992, pp. 136–143.
44. Ulusoy, Ö., "Concurrency Control in Real-Time Database Systems," Technical Report, UIUCDCS-R-92-1762, University of Illinois at Urbana-Champaign, 1992.