

Real-Time Replication Control for Distributed Database Systems: Algorithms and Their Performance

Sang H. Son and Fengjie Zhang
Computer Science Department
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Schedulers for distributed real-time database systems must satisfy timing constraints of transactions and preserve data consistency. This paper presents a replication control algorithm, which integrates real-time scheduling and replication control. The algorithm adopts a majority consensus scheme for replication control and employs epsilon-serializability, a correctness criterion which is less stringent than conventional one-copy serializability. The performance of the algorithm is evaluated and compared with that of a real-time token-based algorithm.

1. Introduction

In real-time distributed database systems, timeliness of results can be as important as their correctness [Yu94]. The problems related to replication control such as the preservation of mutual and internal consistency become more difficult when timing constraints are imposed on transactions. Transactions must be scheduled to meet the timing constraints and to ensure data consistency. Real-time task scheduling can be used to enforce timing constraints on transactions, while concurrency control is employed to maintain data consistency. Unfortunately, the integration of the two mechanisms is non trivial because of the trade-offs involved. Serializability may be too strong as a correctness criterion for concurrency control in database systems with timing constraints, for serializability severely limits concurrency. As a consequence, data consistency might be compromised to satisfy timing constraints.

In replication control methods, on the other hand, the objective is to provide a high degree of concurrency

and thus faster average response time without violating data consistency [Son87]. Two different policies can be employed in order to synchronize concurrent data access of transactions and to ensure identical replica values: *blocking* transactions or *aborting* transactions. However, blocking may cause priority inversion when a high priority transaction is blocked by lower priority transactions. Aborting the very same lower priority transactions, though, wastes the work done by them. Thus, both policies have a negative effect on time-critical scheduling.

A less stringent, general-purpose consistency criterion is necessary. The new criterion should allow more real-time transactions to satisfy their timing constraints by temporarily sacrificing database consistency to some small degree. *Epsilon-serializability* (ESR) is such a correctness criterion, offering the possibility of maintaining mutual consistency of replicated data asynchronously [Pu91]. Inconsistent data may be seen by certain query transactions, but data will eventually converge to a consistent state. In addition, the degree of inconsistency can be controlled so that the amount of the accumulated error (departure from consistency) in a query can possibly be reduced to within a specified margin.

In this paper, we present a replication control algorithm that allows as many transactions as possible to meet their deadlines and at the same time maintaining the consistency of the replicated data. The algorithm is based on majority consensus approach. Epsilon-serializability is employed as the correctness criterion to guarantee the consistency of the replicated database. We also present results from the performance study of our real-time majority consensus algorithm compared to a different real-time replication control algorithm based on token-based synchronization scheme [Son93].

2. Database Model

A distributed system consists of multiple autonomous computer systems (sites) connected via a communication network. Each site maintains a local database system. In order for transactions to be managed properly and for the results of their execution to be applied consistently to all replicas, a special process called *transaction manager* runs at each site. Given the distributed nature and the increased communication burden of such a data-

This work was supported in part by ONR and CIT.

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95) Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995.
© World Scientific Publishing Co. Pte Ltd

base system, a message server process runs at each site and take care of the communication protocols between its site and all others. Data managers are low-level processes, running one per site, that manage the local database [Son92].

The smallest unit of data accessible to the user is called *data object*. In distributed database systems with replicated data objects, a logical data object is represented by a set of one or more replicated physical data objects. In a particular system, a physical data object might be a file, a page or a record. We assume that the database is fully replicated at all sites.

For the majority consensus algorithm, each data object copy is a read-write copy. It consists of a data value and a time-stamp. The time-stamp of a data object reflects the time-stamp of the transaction that last updates the data object. They are critical for insuring mutual consistency of all the databases and the internal consistency of each local database.

A *transaction* is a sequence of operations that takes the database from a consistent state to another consistent state. Two types of transactions are allowed in our environment: *query* transactions and *update* transactions. Query transactions consist only of read operations that access data objects and return their values to the user. Thus, query transactions do not modify the database state. Update transactions consist of both read and write operations.

Transactions have their time-stamps constructed by adding 1 to the greater of either the current time or the highest time-stamp of their base variables. Base variables are a joint set of its read-set and write-set. We assume that for each transaction, the write-set is a subset of the read-set. Therefore a transaction must first read a data object before writes to it.

In majority consensus approach [Tho79], transactions first read all the data objects in the read-set from its local database, and then go through the voting procedure. Having received a vote request from a transaction, the site manager first checks to see if all the base variables of the transaction are up-to-date. If all the base variables are current, it will vote OK to accept the request. Otherwise, it will reject the request. A transaction can only abort after it has been rejected at one site. The currency of a data object is checked by comparing the time-stamp of the data object in the request's read-set with the time-stamp of the data object in the local database. If these two time-stamps are the same, it means that the transaction has read the current data. If the object in the local database has a larger time-stamp than the one supplied by the request, the site manager can conclude that the transaction has read an outdated value, therefore it must be

aborted. If the time-stamp supplied by the request is more current, the request is made to wait because this means that the update of the data object has not yet applied at this location.

Two transactions conflict if the read-set of one transaction intersects with the write-set of the other transaction. During the voting process, if the transaction asking for the vote conflicts with a pending transaction, the scheduler will vote either PASS or defer the request based on the transaction's priority. Each transaction is assigned a priority, possibly based on its time-stamp. The transaction will be deferred if it conflicts with a lower priority transaction.

After voting, the scheduler checks to see if the request has been accepted. If the vote is OK and the request has accumulated a majority of OK votes, the scheduler accepts the request and notifies all other schedulers. If the vote is PASS, and the request has enough PASS votes to prevent a majority of OK votes, the request will be rejected.

After the request has been resolved, if the request is accepted, the scheduler will start updating the data objects in the transaction's write-set and reject all the transactions deferred because of the request. If the request is rejected, all transactions deferred by it will be reconsidered.

Because the acceptance of a request is communicated among schedulers, the acceptance of an early transaction may arrive after the acceptance of a later transaction. When databases are updated, care must be taken to ensure that the updates of an early transaction do not overwrite the updates of a later transaction. To achieve this, we need to compare the time-stamp of the update with the time-stamp of the data object in the database. If the time-stamp of the update is more current, the update is made normally, otherwise the update is omitted.

3. Epsilon-Serializability

Epsilon-serializability (ESR) is a correctness criterion that enables asynchronous maintenance of mutual consistency of replicated data [Pu91]. A transaction with ESR as its correctness criterion is called an *epsilon-transaction* (ET). An ET is a query ET if it consists of only reads. An ET containing at least one write is an update ET. Query ETs may see an inconsistent data state produced by update ETs. The metric to control the level of inconsistency a query may return is called the *overlap*. It is defined as the set of all update ETs that are active and affecting data objects that the query seeks to access. If a query ET's overlap is empty, then the query is serializable. The overlap of an active query transaction Q can be used as an *upper bound of error* on the degree of incon-

sistency that Q may accumulate. Given that we are interested in how many update transactions overlap with Q more than which transactions those are, the term overlap, in its further usage, will reflect the cardinality of the set of update transactions that conflict with the query ET Q .

Among several replica control methods based on ESR, we have chosen the ordered updates approach [Pu91]. The ordered updates approach allows more concurrency than 1SR in two ways. First, query ETs can be processed in any order because they are allowed to see intermediate, inconsistent results. Second, update ETs may update different replicas of the same object asynchronously, but in the same order. In this way, update ETs produce results equivalent to a serial schedule; these results are therefore consistent.

There are two categories of transaction conflicts that we examine: conflicts between update transactions and conflicts between update and query transactions.

Conflicts between update transactions can be either *RW* conflicts or *WW* conflicts. Both types must be strictly resolved. No correctness criteria can be relaxed here, since execution of update transactions must remain 1SR in order for replicas of data objects to remain identical.

Conflicts between update and query transactions are of *RW* type. Each time a query conflicts with an update, we say that the query overlaps with this update, and the overlap counter is incremented by one. If the counter is still less than a specified upper bound, then both operation requests are processed normally, the conflict is ignored, and no transaction is aborted. Otherwise, *RW* conflict must be resolved by using the conventional 1SR correctness criteria of the accommodating algorithm.

The performance gains of the above conflict resolution policies are numerous. Update transactions are rarely blocked or aborted in favor of query transactions. They may be delayed on behalf of other update transactions in order to preserve internal database consistency. On the other hand, query transactions are almost never blocked provided that their overlap upper bound is not exceeded. Finally, update transactions attain the flexibility to write replicas in an asynchronous manner.

4. Real-Time Replication Control

In this section, we present a real-time replication control algorithm, based on the majority consensus approach. We discuss the use of ESR correctness criteria to control the inconsistency of query transactions and the mechanism for conflict resolution of real-time transactions.

4.1 Controlling Inconsistency of Queries

Queries are only involved in *RW/WR* conflicts. When a query transaction is submitted to the system, the

user may quantify it with the restriction “*required to be consistent*.” Such a characterization means that all possible future *RW/WR* conflicts between this query and update transactions will have to be resolved in a strict (1SR) way. In other words, *consistent queries (CQs)* are treated in the same fashion as update transactions. Values returned by *CQs* are always correct, reflecting the up-to-date state of the respective data objects.

If no consistency constraints are specified explicitly by the user on a submitted query, then the *ESR correctness criterion* is employed to maintain the query’s consistency. The overlap upper bound is computed, and an overlap counter is initialized to zero. Each time the query conflicts with an update transaction over the same data object and the counter is less than the overlap upper bound, the conflict is ignored, the counter is incremented, the query reads the value of the data object in question and proceeds to read the next object. When the overlap counter is found to be equal to the upper bound, current and all subsequent conflicts must be resolved in a strict manner, so that no more inconsistency will be accumulated on the query.

When a query transaction eventually commits, the user is able to determine the degree of correctness of the data values returned. If the query was qualified as a *CQ*, then the user can be confident that the values returned are consistent. For regular query transactions, the private overlap counter is checked. If the counter is still zero, this means that no conflict has occurred throughout the entire execution of the query and the results must again be perfectly accurate. Such a query falls into the *CQ* class. An overlap counter greater than zero indicates that a certain number of conflicts with update transactions remained unresolved; the query had seen some possibly inconsistent states, and might yield some inaccurate data. This last type of query falls into the “*possibly inconsistent*” queries class.

Since arbitrary queries may produce results beyond allowed inconsistency even within its overlap limit, it is important to restrict ET queries to have certain properties that permit tight inconsistency bounds. A first attempt in this approach is proposed in [Ram91]. It is beyond the scope of this paper to deal with such strategies. In the remainder of the paper, we assume that inconsistency bounds can be enforced by the system if necessary.

4.2. Real-Time Issues

The real-time scheduling part of our scheme has three components: a policy to determine which transactions are eligible for service, a policy for assigning priorities to transactions, and a policy for resolving conflicts between two transactions that want to access the same data object. None of these policies needs any more infor-

mation about transactions than the deadline and the name of the data object currently being accessed.

Transactions that are unable to meet their deadlines are immediately aborted. When a transaction is accepted for service at the local site where it was originally submitted, it is assigned a priority according to its deadline. The transaction with the *earliest deadline* has the highest priority. *High priority* is the policy that we employed for resolving transaction conflicts. Transactions with the highest priorities are always favored. The favored transaction, i.e. the winner of the conflict, gets the resources that it needs to proceed.

4.3. Conflict Resolution

Before presenting the conflict resolution scheme of the algorithm, we first outline the token-based approach for comparison. A detailed description of the token-based approach is presented in [Son93]. In token-based approach, a token designates a read-write copy that contains the latest version of a data object. The site which has a token copy of a logical data object is called a token site, with respect to the logical data object. Each data object is associated with two values; the *after-value* and the *before-value*. The system remembers the before-value for the duration of the transaction that performs an update operation on the data object, so that it can be restored if the transaction is rolled back. The transaction managers that have been involved in the execution of a transaction are called the *participants* of the transaction. The *coordinator* is one of the participants which initiates and terminates the transaction by controlling all other participants.

To read a data object X, the coordinator checks first whether its read operation conflicts with the write operation that has already been issued by another transaction on the same data object. In case such a conflict occurs, coordinator waits for the termination of the update transaction or reads the before-value of X and proceeds, depending on whether the conflicting transaction is an older or a younger one respectively.

For a write operation to be processed, the coordinator checks for conflicts with other read or write operations. In case that there is a conflict with an older transaction, current update transaction has to be aborted. If it conflicts with a younger transaction, coordinator will have either to wait for the younger transaction to terminate or will proceed but not terminate before the other transaction terminates. After all conflicts, if any, are resolved the value of the data object to be written is broadcast to all token sites where a token-copy of the data object resides. A logical write operation is considered completed when the update request messages are sent. When a write operation is successfully performed

and the transaction is committed, a new version is created which replaces the previous version of the token copy.

A transaction commits when all token-sites of each data object in the write-set of the transaction have pre-committed, each data object in the read-set of the transaction is read, and there is no active transaction that has seen before-value of any data object in the transactions's write-set.

The majority consensus approach is different from the token-based approach in that all the data objects in a transaction's read-set are read at one time. Hence in this approach, two transactions conflict if the intersection of one transaction's read-set and the other transaction's write-set is not empty. In the token-based approach, even though one transaction's read-set overlaps with the other transaction's write-set, these two transactions may not conflict. Because data objects are read and written one at a time, a transaction might be terminated before the other transaction reads or writes the common data objects. In terms of conflicts between two transactions, the majority consensus approach inherently allows less concurrency than the token-based approach.

A transaction execution in the majority consensus approach is divided into two phases: voting phase and commit phase. A transaction remains in the first phase if it is not accepted. It moves into the second phase after it is accepted, but before it commits. A transaction can only be aborted when it is in the first voting phase. A transaction in the second phase cannot be aborted in order to preserve the database consistency.

In a real-time database system, a transaction is aborted whenever it is found to have missed its deadline. The transaction manager for each transaction periodically checks whether or not the transaction will be able to meet its deadline taking into consideration the fact that the transaction has to update the data objects in its write-set at each database. If the system's current time plus the time to update all data objects in a transaction's write-set is greater than the transaction's deadline, it means that this transaction will not be able to commit before its deadline is reached. In order not to waste any system resources, the transaction will be aborted and removed from the system.

If a transaction is decided to be eligible for system service by the above screening process, it is assigned a priority based on its deadline. For two conflicting transactions, the transaction with more urgent deadline is assigned a higher priority.

Compared with token-based approach, the conflict resolution policy for the majority consensus approach is simpler. Conflicts between two transactions is based on

the intersection of their read-sets and write-sets.

Another reason for the simple conflict resolution policy is that a transaction's time-stamp does not contribute to the resolution of conflicts. The only deciding factor in a conflict is the transaction's deadline. Whereas in the real-time token-based approach, both the transaction's deadline and time-stamp are factors in resolving a conflict.

For non-real-time majority consensus approach, the deadline does not affect the resolution of conflicts. Conflicts are resolved based solely on the transactions' time-stamp. Also notice that the transaction's priority in real-time implementation does not refer to the same thing as that of the non real-time version. In the real-time version, a transaction's priority is based on its deadline; but in non-real-time version, it is based on its time-stamp.

Suppose that transaction T1 is already in the pending queue at site A while it is still in the vote-phase. Transaction T2 requests for a vote at site A and T2 then conflicts with T1. The conflict is resolved as follows:

If T1 has higher priority than T2, T1 is deferred until T2 terminates. If T2 commits successfully, T1 will then be aborted later. If T2 aborts later, T1 will be able to continue its execution.

If T1 has lower priority than T2, T2 is aborted and deferred until T1 terminates. Notice here that we abort T2 and make it wait until T1 finishes. This is because if T2 is restarted right away, it will probably conflict with T1 again, and be aborted again, resulting in a waste of system resources.

4.4 Correctness

To establish the correctness of the algorithm, we argue that the algorithm is a special case of the quorum consensus algorithm [Her86]. The correctness proof of the quorum consensus algorithm is provided in [Ber87]. Another proof for the majority consensus algorithm is in [Tho79].

In the quorum consensus algorithm, each read (or write) operation on a data object is translated into reads (or writes) on copies of the data object in some read (or write) quorum. The number of copies in a read quorum R and write quorum W must satisfy that both $2*W$ and $(R+W)$ are greater than the number of total copies. Each write operation updates all the copies in its write quorum. Each read operation reads each copy of its read quorum and returns the most up-to-date copy.

In order to show that the majority consensus algorithm is one type of quorum consensus algorithm, we need to show the majority consensus algorithm satisfies the following three conditions.

(1) Read and write operations satisfy the requirement for read and write quorums.

(2) Each write operation updates all the copies in its write quorum.

(3) Each read operation returns the latest copy in its read quorum.

Condition (1): because every query and update transaction has to get a majority of the OK votes in order to be accepted, the read and write quorum must satisfy: $R = W = \lceil \text{No. of Sites}/2 \rceil$. Also in MCA, each data object is fully replicated at all other sites. So number of sites equals number of copies of each data object. So we have $2*W = (R+W) > (\text{No. of Total Copies})$.

Condition(2): this is satisfied because of the MCA time-stamp generation rule.

$$T = 1 + \max(\text{time}, \max(\{T_b\}))$$

"time" is the time obtained from the global clock.

$\{T_b\}$ is the time-stamps for the transaction's base variables.

So when the updates are applied, each data object will have the updating transaction's time-stamp which is greater than its original time-stamp.

Condition (3): this is guaranteed by the voting rule. The voting rule requires that each site checks the time-stamps of the transaction's base variables to see if any of the base variables have been updated at that site since the transaction was constructed by its initial site. If any of the base variables have been updated, the request will be rejected. This will make sure that once a request is accepted, the values returned by the read operations in the request will be the most up-to-date.

5. Performance Results

In this section we compare the above real-time replication control scheme (RTS) with the respective conventional non real-time scheme (NRTS) on which our algorithm is based. We present a number of performance experiments for the two approaches under various assumptions about the transaction load, the percentage of update transactions in the total number of transactions submitted to the system during the simulation period, and the database size.

A real-time distributed database *prototyping environment* [Son92] was used to build the simulation program. The environment provides the user with multiple threads of execution and guarantees the consistency of concurrently executing processes. Several requests can be submitted at the same time, and many read/write operations take place simultaneously at different sites.

5.1 Metrics and Parameter Settings

The performance metric employed is the percentage of transactions that missed their deadlines (*% missed*) [Abb92] in the total number of transactions that were submitted to the system during the simulation period:

$$\% \text{ missed} = \frac{\text{tardy transactions}}{\text{transactions arrived at the system}} \times 100$$

In case their time slack is not sufficient for their minimum execution time requirements, transactions are permanently removed from the system and the *tardy transactions counter* is incremented.

Transactions are ready to execute as soon as they are submitted (i.e., release time equals arrival time), and the time between transaction arrivals is *exponentially* distributed. The data objects accessed by transactions are chosen *uniformly* from the database.

The deadline assigned to every incoming transaction is given by the formula [Abb92]:

$$\text{deadline} = \text{arrival_time} + \text{execution_time_estimate} + \text{slack_time}$$

The *execution time estimate* is computed as the time that the transaction would need to read the local values of the data objects in its read-set and write the new values of the data objects in its write-set in all of the token-sites, provided that no conflicts occur throughout its whole execution. The execution time estimate represents an average over the minimum times transactions take to complete. The *slack time* is controlled by two parameters, *dead1*, *dead2*, which set a lower and an upper bound, respectively, on its values. The slack time is chosen uniformly from the range specified by these bounds.

In actuality, deadlines can be unreasonable or impossible to meet. In the experiments we performed, we tried to generate deadlines that are neither very easy to meet (in which case all transactions would then trivially be able to meet them) nor very strict (in which case every transaction would definitely miss them). This made it easier to distinguish which algorithm performs the best under identical workload scenarios.

Certain parameters that determine system configuration and transaction characteristics remain fixed throughout the experiments: the *database size* (1000 data objects), the *transaction size* (12 data objects), the *computation cost* per update (8 msec), the *I/O cost* (20 msec), the *percentage of objects* to be updated in each transaction (40%), the *abort cost* for each transaction (19 msec), and the *overlap factor* (0.03, i.e. 3% of the queries return possibly incorrect data). These values are not meant to model a specific distributed database application, but were chosen as reasonable values within a wide range of possible values. In particular, we want transactions to

access a relatively large fraction of the database (1.2%) so that conflicts occur more frequently.

Parameters used as independent variables in one-variable functions describing the *% missed deadlines* performance metric are the *mean inter-arrival time* of transactions (varying between 10msec and 80msec), the *database size* (varying from 200 to 1,000 data objects), and the *percentage of read-only transactions* submitted to the system (varying from 10% up to 90%).

Three major categories of experiments were performed. Each category covers 2 different degrees of distribution, namely distributed databases consisting of 5 and 10 sites. For the 5 sites configuration, cases of 1, 3, and 5 token-sites per data object were examined, while for the 10 sites configuration, cases of 3, 7, and 10 token-sites per data object were considered.

5.2. Comparison of Real-Time and Non-Real-Time Majority Consensus Approach

As we can see, from figures 1 to 6, in all three groups, the real-time version performs significantly better than the non-real-time version in terms of missed deadlines. This can be explained by three reasons. First, ESR is used as the correctness criterion, which implies that the read-only transactions can allow a specified number of inconsistent reads. Second, the conflict resolution rule is sensitive to transaction deadlines thus allowing more transactions to finish before their deadlines. Third, aborted transactions are deferred until the transaction that caused the conflict is terminated, whereas in the non-real-time case, aborted transactions wait for a certain period and then restart again. Because the waiting period is short compared to the transaction execution time, when a transaction restarts, it usually conflicts with the same transaction again. In the real-time case, this is avoided by making the aborted transaction defer, thus saving the unnecessary cost of aborting.

5.3. Comparison of Token-Based Approach and Majority Consensus Approach

Here we are only interested in the comparison of the two real-time implementations. From figures 1 and 2, we noticed that when the inter-arrival time is short, the token-based approach performs better than majority consensus approach. As the inter-arrival time increases, the percentage of missed deadlines for the majority consensus approach decreases more rapidly than the token-based approach. Similar phenomena can also be observed in figures 3 and 4. When the percentage of read-only transactions is very low, the token-based approach has fewer number of transactions miss their deadlines. As the percentage of read-only transactions increases, the majority consensus algorithm starts to out-

perform the token-based algorithm. Because both the transaction inter-arrival time and the percentage of read-only transactions are indicators of the distributed system conflict level, we can conclude that the token-based algorithm performs better when there are more conflicts, while the majority consensus algorithm performs better as the number of conflicts decreases. This is shown in two groups of experiments where the conflict level of the distributed database system is affected by the inter-arrival time and percentage of read-only transactions.

In the token-based approach, some of the RW conflict can be resolved by one transaction reading the before value, thus increasing the concurrency level. But in order to achieve high concurrency, each transaction has to maintain precedence constraints and use a more complicated commit protocol. In majority consensus approach, two transactions can not execute concurrently if they conflict and therefore the concurrency control method is very simple. Also the majority consensus algorithm uses a simple commit protocol. A transaction can commit once it receives a majority of votes and updates the data objects in its write-set. At high conflict level, with more transactions competing for resources, the approach which allows more concurrency (i.e., the token-based approach), performs better. As the conflict level decreases, the approach which incurs lower communication cost or uses less complex concurrency control (i.e., the majority consensus approach), has an advantage.

6. Concluding Remarks

In this paper we have presented a synchronization scheme for real-time distributed database systems. The algorithm is based on the *majority consensus approach*, in which two additional components are built. The first is a set of *real-time constraints* that each transaction has to meet. A separate priority scheme is employed to reflect the demand of a transaction to finish before its deadline. The second component is the *ESR correctness criteria* with which query transactions have to comply. Instead of applying 1SR to all transactions, 1SR is applied only to updates, and queries are left free to be interleaved with updates in a more flexible way.

By relaxing the consistency criteria for query transactions, queries and updates hardly ever have to abort or block each other due to conflicts between them. As an immediate consequence of this, more transactions may terminate successfully before their deadlines expire. Simulation experiments performed using the distributed database prototyping environment support the above theoretical argument for increased performance.

We have compared the performance of two real-time replication control algorithms. Although the token-based

algorithm allows a higher degree of concurrency, it requires a more complex algorithm in terms of data structures the system has to maintain, and incurs higher overhead. The majority consensus algorithm is a simple and elegant algorithm. Although it does not allow as much concurrency as the token-based algorithm, it is very efficient when the system conflict level is low due to its simplicity.

REFERENCES

- [Abb92] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation", ACM Trans. on Database Systems, vol. 17, no. 3, Sept. 1992, pp 513-560.
- [Ber87] P.A.Bernstein, V.Hadzilacos, and N.Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Co., 1987.
- [Her86] M.Herlihy, "A Quorum-Consensus Replication Method for Abstract Data Types", ACM Trans on Database Systems, Vol. 4, No. 1, pp. 32 - 53, 1986.
- [Pu91] C.Pu, and A.Leff, "Replica Control in Distributed Systems: An Asynchronous Approach", ACM SIGMOD Conference, Denver, May 1991.
- [Ram91] K. Ramamritham and C.Pu, "A Formal Characterization of Epsilon Serializability", Tech. Rep., Dep. of Computer and Information Science, University of Massachusetts, Massachusetts, December 1991.
- [Son87] S.H.Son, "Synchronization of Replicated Data in Distributed Systems," Information Systems, vol. 12, no. 2, pp 191-202. 1987.
- [Son92] S.H.Son, "An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems," Journal of Systems Integration, vol. 2, no. 1, February 1992, pp 67-90.
- [Son93] S. H. Son and S. Kouloumbis, "A Token-Based Synchronization Scheme for Distributed Real-Time Databases," Information Systems, vol. 18, no. 6, Dec. 1993, pp 375-389.
- [Tho79] R.H.Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Trans on Database Systems, Vol. 4, No. 2, pp. 180 - 209, June 1979.
- [Yu94] P. Yu, K. Wu, K. Lin, and S. H. Son, "On Real-Time Databases: Concurrency Control and Scheduling," Proceedings of IEEE, Special Issue on Real-Time Systems, January 1994, pp 140-157.

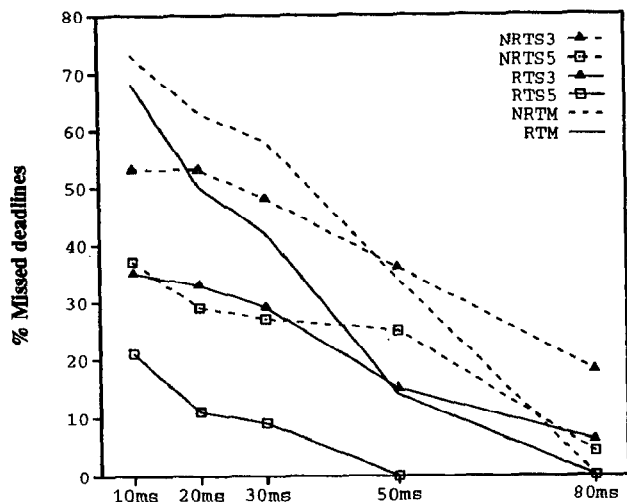


Figure 1: SITES=5, Interarrival time.

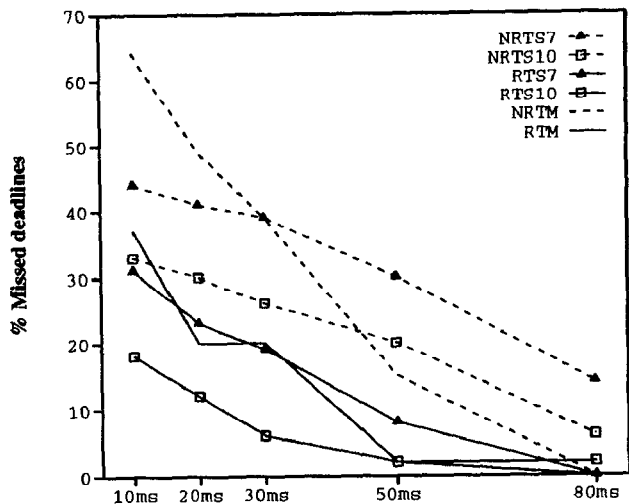


Figure 2: SITES=10, Interarrival time.

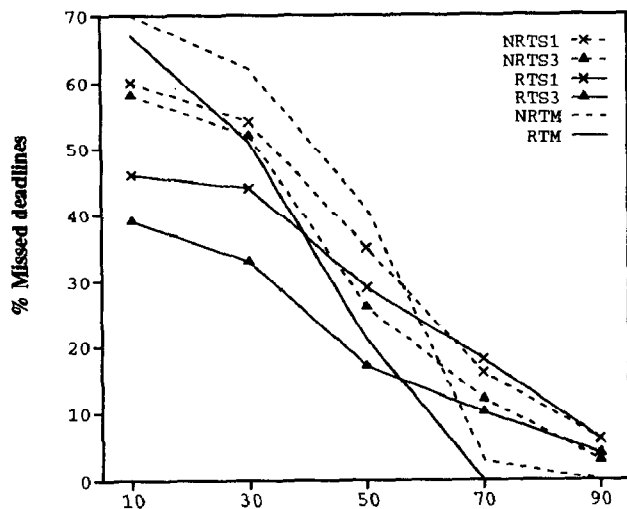


Figure 3: SITES=5, Read only trx %.

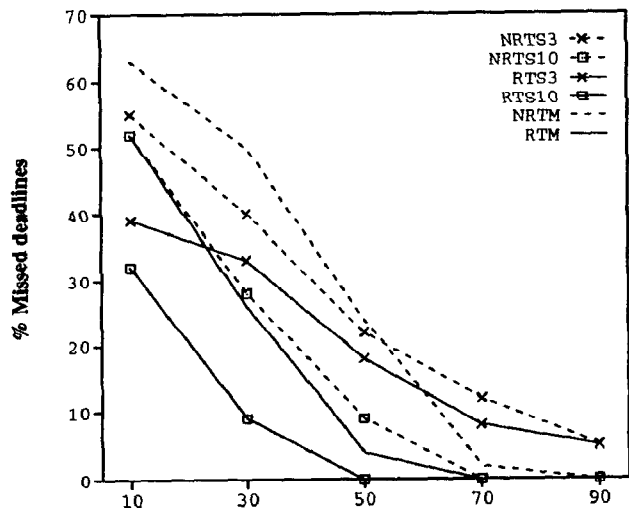


Figure 4: SITES=10, Read only trx %.

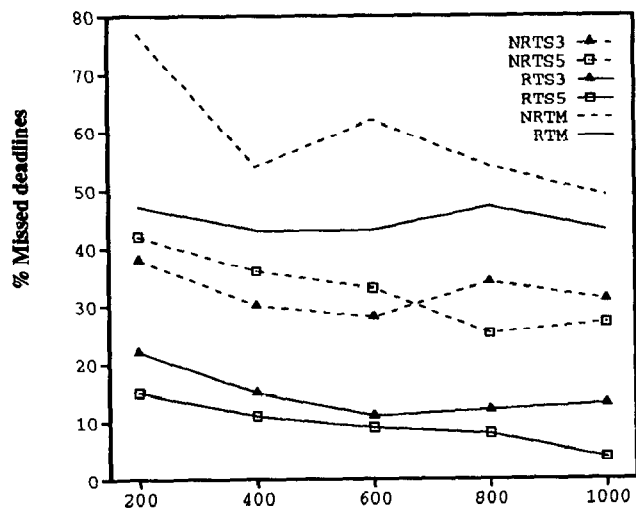


Figure 5: SITES=5, Database size.

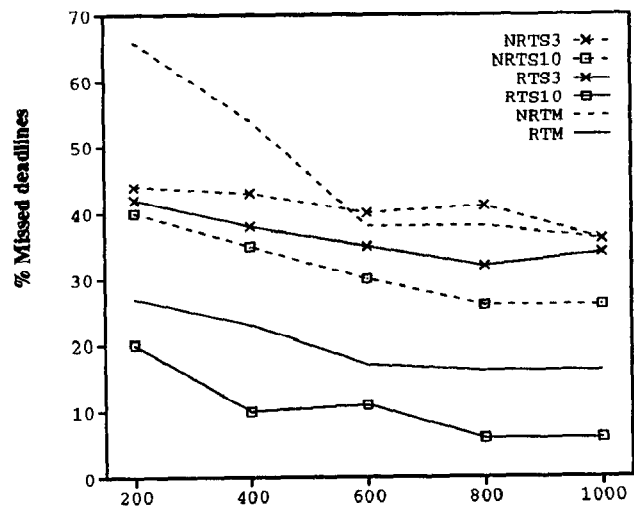


Figure 6: SITES=10, Database size.