A Real-Time Synchronization Scheme for Replicated Data in Distributed Database Systems

Sang H. Son
Spiros Kouloumbis

Computer Science Report No. TR-91-12
May 21, 1991

A Real-Time Synchronization Scheme for Replicated Data in Distributed Database Systems

Sang H. Son
Spiros Kouloumbis

Computer Science Department
University of Virginia
Charlottesville, VA 22903

ABSTRACT

The design and implementation of time-critical schedulers for real-time distributed replicated database systems must satisfy two major requirements: transactions must be able to meet the timing constraints associated with them, and mutual and internal consistency of replicated data must be preserved. In this paper, we propose a new replication control algorithm, which integrates real-time scheduling and replication control. The algorithm adopts a token-based scheme for replication control and attempts to balance the criticality of real-time transactions with the conflict resolution policies of that scheme. In addition, the algorithm employs Epsilon-serializability (ESR), a new correctness criterion which is less stringent than conventional one-copy-serializability, to guarantee the robustness of the scheme. Furthermore, the algorithm is flexible and very practical, since no prior knowledge of the data requirements or the execution time of each transaction is required.

This work was supported in part by the Office of Naval Research under contruct number N00014-88-K-0245, by the Naval Ocean Systems Center, and by the Fedelal Systems Division of IBM Corporation.

1. Introduction

In *Real-time Distributed Database Systems* (RTDDBS), transactions must be scheduled in such a way that they complete before their corresponding deadlines expire. In other words, the *timeliness* of results can be as important as their *correctness*. The main motivations behind the design and implementation of distributed systems include increased availability of data, improved fault tolerance, enhanced performance, and distributed workload [Son87b]. However, the problems related to replication control in those systems, such as the preservation of mutual and internal consistency of replicated data, the concurrency control with the inherent communication delays, become even more pressing and hard to solve when timing constraints are imposed on transactions.

Transactions in RTDDBS can be categorized as *hard* or *soft* transactions. A real-time transaction is said to be hard if its timing constraints must be guaranteed. A hard transaction cannot miss its deadline or else the results will be useless. On the contrary, soft real-time transactions have timing constraints but they are not guaranteed to make their deadlines. The usefulness of their results just decreases after their deadlines are missed.

The goal of scheduling in RTDDBS is twofold: to meet the *timing constraints* and to ensure that the replicas remain *mutually consistent* [Lin89, Son90c]. 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. Moreover, current database systems do not schedule transactions to meet response-time requirements, and they commonly lock data to assure consistency. Unfortunately, locks and time-driven scheduling are basically incompatible: less urgent transactions may block more urgent transactions, leading to timing requirement failures.

In real-time scheduling, tasks are assumed to be independent, and the time spent synchronizing their access to shared data is assumed to be negligible compared with execution time. Data consistency is not a consideration in real-time scheduling, and hence the problem of guaranteeing the consistency of shared data is ignored. Knowledge of resource and data requirements of tasks is also assumed to be available in advance.

In replication control methods, on the other hand [Bre82, Her86, Tho79], the objective is to provide a high degree of concurrency and thus faster average response time without violating data consistency [Ber84]. 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.

Conventional coherency control algorithms are synchronous, in the sense that they require the atomic updating of some number of copies [Ber81]. This leads to reduced system availability and decreased throughput as the size of the system increases. On the other hand, asynchronous coherency control methods that would allow more transactions to meet their deadlines suffer from a basic problem: the system enters an inconsistent state in which replicas of a given data object may not share the same value. Standard correctness criteria for coherency control such as the *1-copy serializability* (1SR) [Ber87] are thus hard to attain with asynchronous coherency control.

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 [Pu91b]. Inconsistent data may be seen by certain query transactions, but data will eventually converge to a consistent (1SR) state. Additionally, 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.

The goal of our work is to design a replication control algorithm that will allow as many transactions as possible to meet their deadlines and at the same time maintain the consistency of replicated data [Son90b] in the absence of any *a priori* information. Our algorithm, which enjoys this property, is based on a token-based synchronization scheme for replicated data in conventional distributed databases. Real-time scheduling features are developed on top of this platform. Epsilon-serializability is employed as the correctness criterion that guarantees the robustness of the scheme.

The remainder of the paper is organized as follows. Section 2 describes the distributed system environment and transaction model adopted by the algorithm. Section 3 provides the reader with more background information on ESR and presents a mechanism for controlling query inconsistency within certain bounds. Section 4 presents the real-time scheduling policies that will be integrated with the replication control scheme. Section 5 discusses the new scheme as a whole and describes the resolution policies for the various conflicts that may occur among transactions. Section 6 presents some simulation experiments on the efficiency of the new scheme. Finally, section 7 summarizes the advantages and disadvantages of the algorithm.

2. Database Model

Before presenting either the theoretical background or the feasible implementation of our real-time replication scheme, we need first to introduce the organization of the underlying distributed database system. A simplified model of a distributed system is presented, and a short description of how transactions are processed by the system is given.

2.1 Distributed System Environment

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. A scheduler process at each site implements the *divergence control* (as opposed to concurrency control) mechanism presented in the following section. Given the distributed nature and the increased communication burden of such a database system, message server processes run at different sites and take care of the communication protocols between their site and all others. Data managers are low-level processes, running one per site, that manage the local database [Son90a].

The smallest unit of data accessible to the user is called *data object*. A data object is an abstraction that does not correspond directly to a real database item. 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. *Read* and *Write* are the two fundamental types of logical operations that are implemented by executing the respective physical operation on one or more copies of the physical data object in question. A *token* designates a read-write copy. Each logical data object has a predetermined number of tokens, and each token copy is the latest version of the 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. In order to control the access to data objects, the system uses timestamps. 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. Copies without tokens go through the *copy actualization phase*, if necessary, in order to satisfy the consistency constraints of the system [Son87a].

When a transaction performs a write operation to a physical data object, there are two values that are associated with the data object; the *after-value* (the new version) and the *before-value* (the old version). The system remembers the before-value for the duration of the transaction so that it can be restored if the transaction is aborted.

Because the before-value is available during the transaction processing, it is natural to ask if concurrency can be improved by giving out this value [Ste81]. For example, if the transaction T_1 has been given a permission to write the new value of a data object and the transaction T_2 requests to read the same data object, then it is possible to give T_2 the before-value of the data object, instead of making T_2 wait until T_1 is finished. However, an appropriate control must be exercised in doing so, otherwise the database consistency might be violated. In the example above, assume that T_1 has written a new value for two data objects X and Y, and T_2 has read the before-value of X. T_2 wants to read Y also. If T_2 gets the after-value of Y created by T_1, there is no serial execution of T_1 and T_2 having the same effect because in reading the before-value of X, T_2 sees the database in a state before the execution of T_1, and in reading the after-value of Y, T_2 sees the database in a state after the execution of T_1.

2.2 Transactions

A *transaction* is a sequence of operations that takes the database from a consistent state to another consistent state. It represents a complete and correct computation. 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. They execute a sequence of local computations and update the values of all replicas of each associated data object. Our real-time replication control approach pays no attention to the local computations; scheduling decisions are made on the basis of the data objects in the read or write-set of the transaction as well as on the criticality of the transaction [Sha88]. The criticality of a transaction at a given point in time depends on the transaction's deadline, and the system priority of the transaction. The system priority is a measure of the value to the system of completing the transaction within its timing constraints [Car89].

Transactions arriving at the system are assumed to be non-periodic. A globally *unique timestamp* is generated for each transaction [Lam78]. Timestamps consist of two fields: the clock value and the node number. When a timestamp is assigned to an incoming transaction, its node's identification number is appended to its local clock value. Each time a transaction is aborted and resubmitted, a new timestamp value is assigned to it. If a transaction T_1 has a smaller timestamp than another transaction T_2, we say that T_1 is the older transaction and T_2 is the younger one.

We assume no *a priori* knowledge of which or how many data objects are going to be accessed by each individual transaction. However, we assume that the average length of query and update transactions are known in order for the algorithm to be applied correctly. Moreover the *run-time estimate* of each transaction is assumed to be unknown. Transactions that miss their deadlines are immediately aborted, so that they do not prevent other non-tardy transactions from committing within their timing constraints.

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.

Read or write operations of the same transaction are executed one by one in a serial fashion. A transaction is said to be *pending* on an object x each time a read/write request on x issued by this transaction is accepted. Each read and write carries the timestamp of the transaction that issued it, and each copy carries the timestamp of the transaction that wrote it. A conflict occurs when a transaction issues a request to access a data object for which other transaction has previously issued a request to access, and furthermore at least one of these requests is a write request. If both requests are read requests, both will be granted. There are three kinds of transaction conflicts [Ber87]:

(1) Read-write (RW) conflict: A transaction requests to read a data object for which other transaction already issued a write request.
(2) Write-read (WR) conflict: A transaction requests to write a data object for which other transaction already issued a read request.

(3) Write-write (WW) conflict: A transaction requests to write a data object for which other transaction already issued a write request.

In each case, we say that the transaction requesting the new access has caused a conflict. Let T_1 be the transaction which already issued an access request, and T_2 cause the conflict. For each token copy of X, conflicts are resolved as the following [Son87a]:

(1) RW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then it reads before-value of X.
(2) WR conflict: If T_2 is younger than T_1, then its write request is granted with the condition that T_2 cannot commit before the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.
(3) WW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.

The coordinator of an update transaction maintains the *before-list* (BL), a list of transactions which read the before-value of any data object in its write set, and the *after-list* (AL), a list of transactions which write the after-value of any data object in its read set. The BL and AL are used during the commitment phase of every update transaction.

When a transaction T_2 reads the before-value of a data object x locked by T_1, the token-site which gives the before-value, conveys the identifier of T_2 to the coordinator of T_1. Hence, the identifier of T_2 is inserted in the before-list of T_1, which stores all the transactions that read the before-values of any data object in T_1's write-set. The transaction manager at the read-only site of T_2 also conveys the identifier of T_1 to the coordinator of T_2. Actually, the identifier of T_1 is inserted in the after-list of T_2, which stores all the transactions that write the after-value of any data object in T_2's read-set.

When a transaction terminates (either commits or aborts), the coordinator of the terminating transaction must inform the coordinator of each transaction in its AL about the termination by sending Termination Messages (TM). On receiving a TM from the coordinator of a transaction in its BL, the coordinator of the active transaction removes the identifier of the terminating transaction (sender of the TM) from the BL. A transaction can commit only when its BL is empty. By this way, we prevent non-serializable execution sequences to occur.

Update transactions have their own *private workspace* [Ber85] where they initially apply their write operations. Update transactions commit by employing a two-phase protocol. In the first phase (*vote-phase*), an update transaction sends an update message to each token-site of every data object in its write-set. The transaction waits until it gets a response from all the token-sites for each data object. If all token-sites vote YES, then the transaction enters the second phase (*commit phase*). It sends the actual value of each data object to be written to the respective token-sites. Update messages to non token-sites can be scheduled after commitment. Therefore, a temporary and limited difference among object replicas is permitted: these replicas are required to converge to the standard 1SR coherency as soon as all the update messages arrive and are processed An update transaction which executes its commit phase can never be aborted, even if it potentially conflicts with another transaction.

Consider a read operation on a data object x. If the local copy of x has *timestamp* > *timestamp*(T_i) then the local value is returned. Otherwise, an *Actualization Request Message* (ARM) is sent to any available token-site to actualize the read-only copy. At the token-site, an ARM is treated the same as a read request, and the current version of the data object will be returned. Nevertheless, query transactions are not always guaranteed to return accurate results. Actually, query transactions fall into three different categories as far as the correctness of their response is concerned:

- *Required consistent queries.* Queries are specified as such when they are first submitted by the user, and they are always guaranteed to return coherent data;
- *Consistent queries.* Their final output is correct regardless of any requirement by the user;
- *Possibly inconsistent queries.* In the case of such a query, there exists a small possibility that returned values of a replicated data object might reflect an inconsistent state of the database.

3. Epsilon-Serializability

Serializability (SR) is the standard notion of correctness in transaction processing. SR in distributed environments maintains database consistency by requesting sites to agree atomically to update objects in a given order. Thus, SR transactions are inherently synchronous. However, implementing SR, requires tight coupling of participant sites and causes system availability and throughput to decrease as the system size increases. More deadlines are likely to be missed when applied to RTDBS. Previously proposed asynchronous coherency control methods permit the system to enter an inconsistent state in which replicas of a given object might not share the same value [Lin89]. Even if distributed systems are willing to pay the price of some inconsistency in exchange for the freedom to do asynchronous updates, they still require that the degree of inconsistency be bounded, and that the system guarantee convergence to a standard notion of "correctness".

3.1 Definitions

Epsilon-Serializability (ESR) is a correctness criterion which offers the possibility of asynchronously maintaining mutual consistency of replicated data [Pu91a, Pu91b]. ESR allows inconsistent data to be seen, but requires that data will eventually converge to a consistent (1SR) state. The degree of inconsistency can be controlled so that the amount of error can be reduced to a specified margin. A distributed system which supports ESR permits temporary and limited differences among object replicas: these replicas are required to converge to the standard 1SR coherency as soon as all the update messages arrive and are processed. These systems benefit from the increased asynchrony allowed under ESR resulting from the *controlled inconsistency of queries* and from the use of *operation semantics* (e.g. commutative operations). The algorithm presented in this paper makes use only of the controlled relaxation of query consistency. Operation semantics are very application-dependent, and since our scheme is meant to be a general-purpose, real-time, replication control scheme, we make no further reference to semantic information.

The high-level interface that encapsulates the ESR abstraction is called *epsilon-transaction* (ET). An ET is a sequence of operations on data objects. These operations are divided into two classes: reads and writes. An ET containing only reads is a query ET (denoted by Q^{ET}) and an ET containing at least one write is an update ET (denoted by U^{ET}). An U^{ET} preserves data consistency. In other words, if the objects modified by an U^{ET} are initially in a consistent state, then after the ET finishes (without interference from operations outside the ET) the objects will again be in a consistent state. While conventional SR transactions guarantee the property of atomicity (i.e. *non-interference* and execution in an *all or none* fashion), E-transactions are assumed to be *approximately atomic* (i.e. updates are assumed to be atomic within certain time lags; data object replicas have the same value when updates complete).

Just as concurrency control is the mechanism for maintaining data consistency in SR-logs, *divergence control* is the mechanism for ensuring correctness of ESR-logs. *ESR-logs* consist only of query ETs and update ETs. An ESR-log is correct if after deleting all query ETs from the log, the remaining update ETs form a log equivalent to a serial log. An example of an ESR log is:

$$R_1(a)\ W_1(b)\ W_2(b)\ R_3(a)\ W_2(a)\ R_3(b)$$

Even though $U_2^{ET}=W_2(b)W_2(a)$ and $Q_3^{ET}=R_3(a)R_3(b)$ are not SR, the deletion of Q_3^{ET} results in an SR log (actually a serial log) formed by U_1^{ET} and U_2^{ET}. As a result, the above log has the ESR property.

Consequently, query ETs can be freely interleaved with other ETs (queries or updates). Query ETs may see an inconsistent object state produced by update ETs. This property does not disturb data consistency, since query ETs do not change object state; it does increase concurrency however, because it increases the number of allowed interleavings. The metric that allows us to control the amount of inconsistency a query may accumulate by arbitrarily interleaving with update transactions is the *overlap*. We define the overlap of a query ET as the set of all update ETs that had not finished at the first operation of the query ET, plus all the update ETs that started during the query ET. The term *update ETs* refers to update transactions actually affecting data objects that the query transaction seeks to access. If a query ET's overlap is empty, then the query is serializable.

We reiterate here the distinction between replica control, which ensures asynchronous mutual consistency under ESR, and traditional coherency control, which ensures synchronous mutual consistency under 1SR. Also, we distinguish replica control from the maintenance of system internal consistency, termed divergence control. This distinction is analogous to the distinction between coherence control (replicas of a "single" data object) and concurrency control (system internal consistency).

We therefore restate the ESR model in terms of replica control in a distributed system. First of all, we are only concerned with ETs that carry information about replicated data. At each site, an ET is represented by a message set or MSet. Query ETs use query MSets to read the values of an object's copy. An update MSet is a set of replica maintenance operations which propagates updates to object replicas. That is, when an update is originated in a client node, the results of the update are

propagated to the replicas in MSets. Each local system is responsible for applying its MSet and preserving internal consistency. Note that the propagation of MSets to each site is asynchronous. We assume the system maintains the unprocessed MSets in some stable storage. Each MSet is stored as an element in a stable queue. Due to the asynchronous propagation of MSets, replicas of a "logical" object can differ at any given moment. This is a source of inconsistency seen by the query ETs. A key observation, however, is that under ESR all replicas converge to the same 1SR value when the update MSets queued at individual sites are processed, and the system reaches a quiescent state.

The known replica control methods can be classified into two families. The first termed "*forward methods*", prevents inconsistency by restricting certain properties of transactions and the system. For example, if update ETs contain only *commutative* operations, then the system supports ESR, since update operations can be reordered into a serial schedule. Another example of replica control is *ordered updates*, which maintains the processing order of update ET operations. Ordered updates maintain system consistency, and query ETs can be processed in any order to increase concurrency. *Read-independent timestamped updates* also preserve ESR, because update ETs can be scheduled in any order, and query ETs are scheduled more or less freely, depending on consistency and concurrency trade-offs.The second family of replica control, termed "*backwards methods*", is based on compensation operations. The replicated system may optimistically allow operations to proceed in parallel. If inconsistencies are detected later, then the system rolls them back with compensation operations or compensation ETs.

In summary, the forward methods assume the updates have been *committed* and are being propagated through a reliable communication mechanism, while the backwards methods supply some recovery mechanism in case the updates are *aborted.*

3.2 Query Overlap Considerations

The overlap of an active query transaction Q can be used as an *upper bound of error* on the degree of incoherence 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. More formally, query Q's overlap is described as follows:

$$Overlap[Q] = \| \{U_i \,/\, U_i \text{ update trans} \wedge U_i \text{ active} \wedge \text{write-set}(U_i) \cap \text{read-set}(Q) \neq \emptyset \} \|$$

Suppose we have a database of A distinct data objects, and that query transactions read m data objects on the average, and possibly conflict with update transactions that update n data objects on the average. Given that p is the maximum percentage of inconsistent query transactions allowed in the system, the upper bound k for the overlap of such queries is:

$$k = \left\lfloor \frac{p}{1 - \frac{(A-m)!\,(A-n-1)!}{A!\,(A-m-n-1)!}} \right\rfloor$$

There are several reasons why we have thus chosen k. First of all, a query could conceivably start at the beginning of a log and finish at the end (i.e. worst case). Such a query would contain as much inconsistency as there are conflicts, resulting in unbounded inconsistency. Even though internal consistency of the database would not have been disturbed, an unbounded amount of inaccuracy would have been accumulated in each query. The user could perceive a totally wrong picture of an essentially consistent database as a consequence of the uncontrolled interleaving between update and query transactions.

On the other hand, the overlap bound k must dynamically adjust to the current database configuration and transaction parameters. It must also be expressed as a function of the degree of inconsistency (percentage of inaccurate query transactions) that the user has decided to tolerate. In other words k must be a function:

$$k = F(db\_size, query\_size, update\_size, \%\_inaccurate\_queries)$$

The user can thereby constrain the desired degree of query inconsistency, for different types of transactions and various degrees of conflicts.

The exact value for the overlap number can be computed as follows. We will compute the maximum allowable overlap of the query Q for a given degree of query inconsistency p. Note that $(100 \times p)$ expresses the percentage of query transactions that might read inconsistent data. We allow a certain query to return incorrect data with at most probability p.

The probability that U_i to accesses n objects different from any of the m objects of Q's read set is: $p_i = \frac{A-m}{A} \times \frac{A-m-1}{A-1} \times \ldots \times \frac{A-m-n}{A-n}$. So the probability that U_i, has common elements with Q (i.e. Q overlapping with U_i) is: $1 - p_i$.

The probability for a query transaction to overlap with an arbitrarily chosen update transaction is: $l_i = 1 - p_i = 1 - \frac{A-m}{A} \times \ldots \times \frac{A-m-n}{A-n} = 1 - \frac{(A-m)!\,(A-n-1)!}{A!\,(A-m-n-1)!}$.

Variable l_i essentially represents the inconsistency probability that a query overlaps with exactly one update transaction ($k = 1$). If k is the maximum overlap, then the equation $\sum_{i=1}^{k} l_i = p$ must hold. We emphasize that we have k distinct update transactions that potentially conflict with the query.

Since we assume that read/update sets are uniformly distributed within the database, we must have $l_i = l \;\; \forall \; i \leq k$, and thus $k \times l = p$. Solving the equation for k after substituting the value for l, we derive the formula presented above.

3.3 E-Transactions Compatibility

Several replica control methods, based on the ESR correctness criteria, are presented in [Pu91a] [Pu91b]. We have chosen the ordered updates approach in order to *relax* the SR criteria and achieve the greatest possible concurrency without corrupting the mutual consistency of replicated data. The ordered updates approach allows more concurrency than strict SR in two ways: first, query E-transactions can be processed in any order because they are allowed to see intermediate, inconsistent results. Second, update E-transactions 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. Furthermore, the amount of inconsistency in the query ETs is bounded by the number of concurrent update ETs with which they may be interleaved.

There are two categories of transaction conflicts that we examine: (1) conflicts between update transactions, and (2) conflicts between update and query transactions. Conflicts between query transactions are not an issue since read operations on the same data objects are assumed always to be compatible and never interfering with each other [Esw76].

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. Besides, ESR requires that the log of update transactions always be serializable.

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 (i.e. the value of k that we derived in the previous section), 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 are sometime 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. Participants receiving update messages for local replicas need not acknowledge their reception back to the coordinator. Replica update order is maintained in the message queues of token-sites where updates are represented as messages ordered in a FIFO manner, and hence in the order that the initiating transaction committed. Thus, update ETs can commit immediately after sending the update requests to remote sites, and will be susceptible to conflicts with other update transactions for the minimum amount of time.

	R_U	W_U	R_Q
R_U	OK	–	OK
W_U	–	–	OK
R_Q	OK	OK	OK

Table 1: Compatibility of E-Transactions

Table 1 shows the resulting *lock compatibility* for ETs when the Ordered Updates approach is adopted by the scheduler for transaction operation conflict resolution.

4. Real-Time Issues

In real-time databases, transactions are characterized by their timing constraints and their data and computation requirements. Timing constraints are expressed through the *release time* and the *deadline*. Computation requirements for transactions are unknown, and no run-time estimate is available for every transaction that enters the system. Neither are data requirements known beforehand, but they are discovered dynamically as the transaction executes. Our goal is to minimize the number of transactions that miss their deadlines [Abb88].

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 lock the same data object. None of these policies needs any more information about transactions than the deadline and the name of the data object currently being accessed.

All transactions which are currently *not tardy* are eligible for service. Transactions that have already missed 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. This policy meshes efficiently with the "not tardy" eligibility policy adopted above, so that transactions that have already missed their deadlines are automatically screened out before any priority is assigned to them. *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 (e.g. data locks and the processor [Car89]). The loser of the conflict relinquishes control of any resources that are needed by the winner. The loser transaction will either be aborted or blocked depending on the relative age of the two conflicting transactions and the special provisions made by the replication control scheme.

5. Replication Control Scheme

In this section, we present the token-based replication control scheme in detail, along with the embedded ESR correctness criteria and the real-time constraints. All possible types of transaction conflicts will be examined, and for each type a step by step resolution will be given. Finally, commitment criteria for the real-time transactions of the system will be presented, and the correctness of the proposed algorithm will be discussed.

5.1 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 identically to update transactions which are not allowed to be interleaved with other transactions in a non-SR way. Values returned by CQs are always correct, reflecting the up-to-date state of the respective data objects.

If no coherence constraints are specified explicitly by the user on a submitted query, then *ESR correctness criteria* are 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. In other words, queries that have reached their overlap limits are treated as update transactions, should a new conflict appear in the future.

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 100% confident that the values returned are coherent. 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 intermediate states, and might yield some inaccurate data. This last type of query falls into the "*possibly inconsistent*" queries class. The probability that such query outputs incoherent data is bounded by the probability p which was used in the calculation of the overlap limit k. Data values can then be referenced with a $[(1-p)\times 100]\%$ confidence in their correctness.

For each query transaction T, we can also provide the ratio of the number of correct values and that of possibly incorrect values read by T – called *correctness level* – by checking the overlap counter of T and the number of data objects read by T. Let m_{ex} be the exact number of data objects

read by T and k_{ex} be the value of the overlap counter of T after T had terminated. The number of possibly incorrect data values read by T is: $m_{ex} \times p_{ex}$, where

$$p_{ex} = k_{ex} \times (1 - \frac{(A - m_{ex})!\,(A - n - 1)!}{A!\,(A - m_{ex} - n - 1)!})$$

is the exact probability that T is inconsistent. The number of possibly correct data values read by T must be equal to $m_{ex} \times (1 - p_{ex})$. We conclude that the correctness level is given by the following formula:

$$correctness\,level = \frac{1 - p_{ex}}{p_{ex}} = \frac{1 - k_{ex} \times (1 - \frac{(A - m_{ex})!\,(A - n - 1)!}{A!\,(A - m_{ex} - n - 1)!})}{k_{ex} \times (1 - \frac{(A - m_{ex})!\,(A - n - 1)!}{A!\,(A - m_{ex} - n - 1)!})}$$

5.2 Conflict Resolution

Mechanisms for conflict resolution between update transactions comprise the core of our scheme. Query transactions need not be considered separately, because, (as it was mentioned in the previous section) queries that are forced to resolve their RW conflicts with update transactions can be treated identically to update transactions. Therefore, in the rest of the section, we use the general term *transaction* when we refer either to a normal update transaction or to a query transaction that has to resolve its conflict.

We use *timestamp ordering* for concurrency control. Each read and write carries the timestamp of the transaction that issued it, and each copy carries the timestamp of the transaction that wrote it. A conflict occurs when a transaction issues a request to access a data object for which another transaction has previously issued a request to access and at least one of these requests is a write request.

We examine three separate categories of conflicts, and for each category we present a table of all possible conflicts between younger and older transactions, and between higher priority and lower priority transactions. Table entries framed in dotted lines represent cases where the real-time considerations must be taken into account during conflict resolution [Hua90].

Before we get into the detailed discussion of conflict resolution in each case, we must clarify what we mean by "conditional abort". Let T_1 be an accepted and successfully processed transaction issuing a write request on data object X. Let T_2 be another transaction attempting to read or write X. In the case that T_1 has to be aborted (possibly for real-time priority reasons or to preserve database consistency), T_1's phase of commitment must first be checked. If T_1 is in the vote-phase, it can be aborted normally. All writes on T_1's private workspace will be discarded, and T_1 will be resubmitted, provided that it has not already missed its deadline. However, if T_1 is in the commit-phase and update messages have already been sent to token-sites, then T_1 cannot be aborted. Con-

sequently, "conditional abort" refers to the action of aborting an update transaction T only in the case where T is still in the vote-phase. If T_1 and T_2 are in conflict and T_1 has entered the second phase of commitment, then T_2 must be aborted in order for the database to remain coherent.

(1) R - W Conflict.

Transaction T_2 requests to read a data object X for which transaction T_1 has already issued a write request. Table 2 shows the various resolution policies.

If T_2 is younger than T_1, then the original scheme requires that T_2 must wait for the termination of T_1 before it reads the value of X. In case T_2 has lower priority than the priority of T_1, the original requirement does not contradict with the priority status and T_2 can wait for T_1 to terminate. On the other hand, if T_2 has higher priority than T_1, this means that T_2 is closer to miss its deadline than T_1 is. Hence, we cannot apply the original rule and force T_2 to wait. Instead, T_2 must read the data object and proceed whereas T_1 has to be conditionally aborted in order for the consistency of data read by T_2 to be preserved.

If T_2 is older than T_1, then the original scheme requires that T_2 read the before-value of X and proceed. T_2 is then inserted in the before-list (BL) of T_1, and T_1 is inserted in the after list of T_2. According to the commitment criteria of the token-based scheme, T_1 will have to wait for T_2 to terminate before it can commit. Such resolution is acceptable when T_2 has higher priority than T_1. However, in the case that T_2 has lower priority than T_1, T_1 is considered to be more urgent and risks the possibility of missing its deadline while waiting for T_2 to terminate. In such a case, we block T_1 only until it is very close to missing its deadline. If such a situation occurs and T_2 has not yet terminated, we unblock T_1 in order to meet successfully its timing constraints, and we conditionally abort T_2 in order to maintain the coherence of data written by T_1.

Priority / *Age*	*T_2 higher priority*	*T_2 lower priority*
T_2 younger	• T_1 is aborted (cond). • T_2 reads.	• T_1 writes. • T_2 waits for T_1 to terminate.
T_2 older	• T_1 writes. • T_2 reads before value. • T_1 waits for T_2 to terminate before it commits.	• T_1 writes. • T_2 reads before value. • T_1 waits for T_2 to commit before it commits. • If T_1 is about to miss its deadline T_2 is aborted (cond), T_1 commits.

Table 2: R-W Conflicts

(2) W - R Conflict.

Transaction T_2 requests to write data object X for which transaction T_1 has already issued a read request. Table 3 shows the various resolution policies in this case.

If T_2 is younger than T_1, then the original scheme requires that T_1 read the before-value of X and T_2 write an after-value of X. Moreover, T_1 is inserted in the BL of T_2, and T_2 is inserted in the AL of T_1. In this way, T_2 will have to wait for the termination of T_1 before it can commit. In the case that T_2 has lower priority than T_1, there is no contradiction between its priority status and its obligation to be blocked. In the case that T_2 has a higher priority than T_1, T_2 will be blocked, but only until it is very close to missing its deadline. If such a situation occurs and T_1 has not yet terminated, T_2 is unblocked and is given the chance to meet its deadline, whereas T_1 is conditionally aborted so that it will not return an outdated value of X.

If T_2 is older than T_1, then the original scheme requires that T_2 be rejected and T_1 read the correct, up-to-date value of X. Aborting T_2 is perfectly justified in the case T_2 has a lower priority, since the more urgent T_1 is favored to proceed towards termination. In the case that T_2 has a higher priority, aborting T_2 would violate the real-time constraints. Therefore, we let T_2 proceed and write a new value for X while T_1 is aborted, since it has seen a value of X that has already become obsolete.

Priority / *Age*	*T_2 higher priority*	*T_2 lower priority*
T_2 younger	• T_1 reads before value. • T_2 writes. • T_2 waits for T_1 to commit before it commits. • If T_2 is about to miss its deadline T_1 is (cond) aborted, T_2 commits.	• T_1 reads before value. • T_2 writes. • T_2 waits for T_1 to commit before it commits.
T_2 older	• T_1 is aborted. • T_2 writes.	• T_1 reads. • T_2 is aborted.

Table 3: W-R Conflicts

(3) W - W Conflict.

Transaction T_2 requests to write data object X for which transaction T_1 has already issued a write request. Table 4 shows the various resolution policies.

If T_2 is younger than T_1, then the token-based scheme requires that T_2 wait for the termination of T_1 before it writes a new value for data object X. Such conflict resolution favors T_1 and is compatible with the situation where T_2 has lower priority than T_1. However, when T_2 has a higher priority, it is not required to wait for the lower priority transaction T_1. Hence, T_2 will proceed, and T_1 will be conditionally aborted in order for the database to remain internally consistent.

If T_2 is older than T_1, then the original approach forces T_2 to be aborted. Note that we are interested only in the most recent value of X, i.e. the value written by the younger T_1 transaction. In the case that T_2 has a lower priority, the above resolution is acceptable, since the higher priority T_1 is favored to proceed. On the contrary, when T_2 has the higher priority, T_2 must be allowed to write its own new value of X, and T_1 must be conditionally aborted for the database to remain coherent with respect to the data object X.

Priority / *Age*	*T_2 higher priority*	*T_2 lower priority*
T_2 younger	• T_1 is aborted (cond). • T_2 writes.	• T_1 writes. • T_2 waits for T_1 to terminate
T_2 older	• T_1 is aborted (cond). • T_2 writes.	• T_1 writes. • T_2 is rejected.

Table 4: W-W Conflicts

5.3 Commitment

The coordinator of a transaction decides to commit when the following conditions are satisfied:

- The transaction must not have missed its deadline;
- Each data-object in the read-set of the transaction is read;
- All the token-sites of each data object in the write-set of the transaction have precommitted (this only applies to update transactions);
- There is no active transaction that has seen before-value of any data object in the transaction's write-set. In other words, the *before-list* of the transaction must be empty (this only applies to update transactions).

5.4 Correctness

A concurrency control algorithm is said to be correct if the same state results when the transactions are processed in a serial fashion. In the distributed database system model we adopted for our scheme, each data object is associated with two values: the before-value and the after-value. The problem of maintaining the consistency of replicated data can be thought as of a special case of a multiversion concurrency control problem where the maximum allowable number of versions for each data object is two. On the other hand, ESR is our correctness criterion. An ESR log is correct if the log derived after removing all the query transactions from the original log is serializable. Therefore the major task of our scheme is to guarantee a serializable execution of update transac-

tions in a multiversion database environment. To accomplish such task, one-copy-serializability (1SR) is required. Before we present the detailed correctness proof, we review some fundamental concepts associated with 1SR. For more formal definitions, see [Ber87].

A read operation issued by transaction T_j on the version X_i of a data object X is denoted as $r_j(X_i)$. By the term *"version"* here we mean either the before-value of X or the after-value of X. In order to know which of these two values is actually accessed, we use the notation $r_j(X_i^{bef})$ and $r_j(X_i^{aft})$ respectively. A write operation issued by transaction T_j is denoted as $w_j(X_j)$, where X_j is the new after-value of X created by T_j. A more analytical notation for a write operation is: $w_j(X_j^{aft})$.

A *log* is a model of execution of transactions, capturing the order in which read and write operations are executed. The precedence relation between transactions is defined as: $T_i \rightarrow T_j$ implies that for some data object X, $r_i(X) \rightarrow w_j(X_j)$ or $w_i(X_i) \rightarrow r_j(X)$ or $w_i(X_i) \rightarrow w_j(X_j)$. Let L be a log over a set of transactions. The serialization graph for L, SG(L), is a directed graph whose nodes are transactions, and there is an edge from T_i to T_j $(i \neq j)$ if $T_i \rightarrow T_j$.

For a multiversion system the serialization graph is extended as follows: given a log L and a data object X, a version order (denoted by <<) for X is any total order over all the versions of X in L. A version order for L is defined as the union of the version orders for all data objects in L. The multiversion serialization graph, MVSG(L), is SG(L) with the following edges added: for each $r_k(X_j)$ and $w_i(X_i)$ in L, if $k \neq i$ and $X_i << X_j$ then include $T_i \rightarrow T_j$, else include $T_k \rightarrow T_i$.

To prove that a log L produced by our scheme is 1SR, we have only to prove that there exists a version order << such that MVSG(L) is acyclic. Edges in MVSG(L) are created due to either WW or RW/WR conflicts between transactions. We examine each of these two cases separately.

• *WW conflict.*

In all but one WW conflict resolution cases, one of the two conflicting transactions, T_1 and T_2 is aborted, so that any dependency between T_1 and T_2 is destroyed. Hence, there is no possibility of having a $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$ edge in the MVSG(L) due to a WW conflict between T_1 and T_2. 1SR is then absolutely guaranteed. In the special case where T_2 is the younger transaction and has lower priority than T_1, T_2 is blocked until T_1 terminates, and then it proceeds to write its own value on X. However, no $T_1 \rightarrow T_2$ edge can be constructed at any time in the MVSG(L), since T_2 resumes execution (and writes X) only after T_1 has terminated and has been totally removed from the MVSG(L). Again, no cycle including T_1, T_2 may be constructed in the MVSG(L) due to the WW conflict between them.

• *RW/WR conflict.*

Let $T_i \rightarrow T_j$ be an edge of MVSG(L). This edge can correspond to a simple read X of T_j from the version written by T_i: $r_j(X_i)$. X_i can be either the before-value of X or the after-value of X. If X_i represents the before-value of X, then we must have the following sequence: $w_i(X_i^{bef}) \rightarrow r_j(X_i^{bef})$. However, such a sequence is infeasible. When T_i originally writes X, it writes an after-value X_i^{aft} of X. For X_i^{aft} to become an X_i^{bef}, another transaction, T_k, must write a new value in X, so that X_k becomes the new after-value X_k^{aft} and the current X_i^{aft} becomes the new before-value X_i^{bef} for X.

Therefore, there exists no transaction T_j that can read X as an X_i^{bef} immediately after X_i is created by T_i. Hence, $w_i(X_i^{bef}) \rightarrow r_j(X_i^{bef})$ is not a legal sequence.

If X_i is the after-value of X, then we must have the sequence: $w_i(X_i^{aft}) \rightarrow r_j(X_i^{aft})$. T_i is supposed to be the transaction that has already been granted a write access on X, and T_j is the incoming one that will be granted a read access on X. From tables 2,3 we note that such an execution sequence is infeasible, since one of the two transactions T_i or T_j will have to be aborted in order for the other one to proceed. Another alternative is that T_j might have to wait for T_i to terminate, which again eliminates the possibility that both transactions will be active after the execution of the $w_i(X_i^{aft})$ operation and when $r_j(X_i^{aft})$ starts to execute. Thus, $w_i(X_i^{aft}) \rightarrow r_j(X_i^{aft})$ is again impossible.

Now consider an edge introduced in L by the relationships among $r_k(X_j)$, $w_j(X_j)$, and $w_i(X_i)$. We have two different possible version orders:

- $X_i << X_j$: X_i must then be X_i^{bef}, and X_j must be X_j^{aft}. By the MVSG(L) generation rule, the edge is $T_i \rightarrow T_j$. However, the scheme maintains the version order in timestamp order. X_i^{bef} must have been created by an older transaction than the transaction that created X_j^{aft}. Thus, we have $timestamp(T_i) < timestamp(T_j)$.
- $X_j << X_i$: X_j is supposed to be X_j^{bef} and X_i be X_i^{aft}. By the MVSG(L) generation rule, the edge is $T_k \rightarrow T_i$. The conflicting operations this time are $r_k(X_j^{bef}) \rightarrow w_i(X_i^{aft})$. In order for transaction T_k to read a before-value X_j^{bef} of data object X while a new value X_i^{aft} of X is created, T_k must be older than T_i. This last statement is derived directly from the functional definition of before/after values. Hence, $timestamp(T_k) < timestamp(T_i)$.

We have shown that all possible edges in MVSG(L) are in timestamp order. Nevertheless, suppose there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in MVSG(L). By the above argument, we will have $timestamp(T_1) < timestamp(T_2) < \ldots < timestamp(T_n) < timestamp(T_1)$. This is impossible. Therefore no cycle can exist in MVSG(L), and the algorithm only produces 1SR logs.

Although internal database consistency is constantly preserved, as shown above, query transactions might occasionally return incorrect values due to the relaxed consistency ESR criteria applied on them. Nonetheless, the user has the privilege of either requesting a strictly correct query, or, in the worst case, knowing whether the returned results are coherent and to what extent.

As far as the real-time correctness of the scheme is concerned, we provide the following informal argument. The higher priority (i.e. closer to its deadline) transaction is never aborted in the majority of the cases where a conflict occurs with a lower priority transaction. In the case where the conflict resolution policy requires some transaction to be aborted, the lower priority transaction is chosen, unless the lower priority transaction is an update one executing its commit phase. In such a case, the higher priority transaction will be forced to abort for database coherency to be maintained. In any other case, where the higher priority transaction T_H might be blocked by a lower priority transaction T_L, T_H will have to wait for T_L, but only until its deadline is about to expire. If such a situation occurs, T_L will be aborted and T_H will resume execution and commit successfully within its timing constraints.

6. Simulation 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* [Son90a] 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. More specifically, for the required degree of parallelism to be achieved, the program makes use of the following four classes of threads. The *main program thread* initializes the database system, triggers the execution of the other three threads, and runs the performance monitor. The *transaction manager threads* represent transactions submitted by the user, each one to be served at a specific site. One *scheduler thread* per site captures the semantics of the implemented concurrency control algorithm and is responsible for handling read/write operations as well as requests for permission to commit a transaction. *Message server threads* control the message traffic between transactions submitted at each site and the respective scheduler at that site, as well as messages exchanged between each local scheduler and schedulers at other sites.

We first discuss the performance metrics of interest and the parameter settings used [Car89] [Car88], and then interpret the performance results, taking into account the individual features of each of the two algorithms.

6.1 Metrics and Parameter Settings

The performance metric employed is the percentage of transactions that missed their deadlines (*% missed*) [Abb88] in the total number of transactions that were submitted to the system during the simulation period:

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

In the above definition, we must emphasize that all transactions will eventually meet their deadlines or become tardy. Transactions that are aborted due to the RW/WW conflicts are automatically resubmitted to their sites provided that it is still possible for them to meet their deadlines. 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. Each time that a transaction finishes executing a time-critical operation, it checks the current time against its deadline. If the deadline was missed during the last operation, the transaction is aborted and removed from the system and the *tardy counter* is incremented, otherwise the transaction continues its execution and schedules the next operation.

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 [Abb90]:

$$deadline = arrival\_time + execution\_time\_estimate + slack\_time$$

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. The high conflict rate allows the real-time mechanism to play a significant role in scheduling performance. The base values of the key simulation parameters for many of our experiments are summarized in Table 5.

Parameter	Setting
Database size (data objects)	1000
CPU cost (msec)	8
I/O cost (msec)	20
Communication cost (msec)	1.0
Abort cost/transaction (msec)	19.0
Transaction size (data objects)	12
Mean inter-arrival time (msec)	30
Read only transactions (%)	40
Data objects to be updated (%)	40

Table 5: Base parameter values.

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/write transactions* submitted to the system (varying the number of read only transactions from 10% up to 90% of the total number of transactions). The values of system configuration and transaction parameters used for simulations are listed in Table 6.

Parameter	Values
Mean inter-arrival time (msec)	5, 15, 30, 60, 100
Read only transactions (%)	10, 30, 50, 70, 90
Database size (data objects)	200, 400, 600, 800, 1000

Table 6: Variable parameter values.

We assume that the database is fully replicated at all sites. A standard, modulo-based formula is used to assign token-sites to a data object. Let *m*, where m is a natural number, denote a data object in our database; let *numsites* be the number of sites; and let *toksites* denote the number of token-sites that must be assigned to data object *m*. Then the following token-sites are going to be assigned to *m*:

$$[\,(m + i)\ MOD\ numsites\,] + 1, \text{ where } i = 0, \ldots, toksites\text{-}1$$

Only aperiodic transactions are submitted to the system. Whenever an executing transaction is aborted, it is automatically resubmitted to the system with a new timestamp. This procedure is repeated for every transaction until the transaction commits or it misses its deadline.

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.

6.2 Transaction Inter-arrival Time

In this experiment we vary the inter-arrival time of transactions from 10msec (for a heavily loaded system), to 80msec (for a non saturated system state). The database has a size of 1,000 data objects and 40% of the transactions submitted to the system are read only.

Figures 1 and 2 show that curves start from a high percentage of missed deadlines and descend until a minimum percentage is reached at the vicinity of 80msec. When the inter-arrival time is 10msec, the system is overloaded by a huge number of transactions striving to access a limited number of data objects (*hot spots*). Consequently, the number of conflicts becomes so large that a lot of transactions will be blocked and eventually miss their deadlines. As the inter-arrival time increases, the number of active transactions per second in the system decreases. Fewer transactions

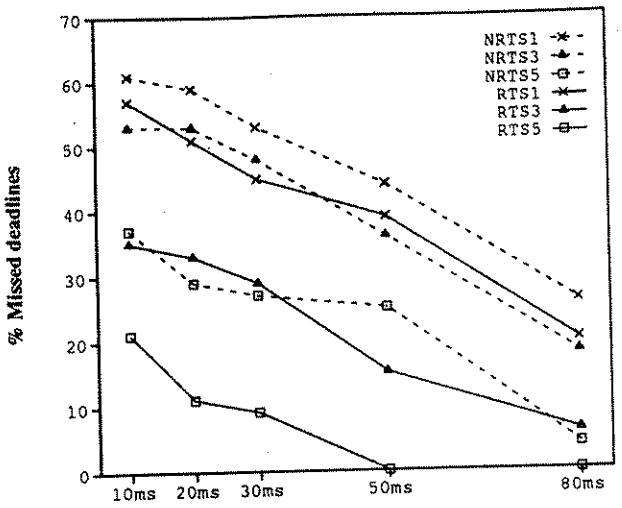

Figure 1: SITES=5, Interarrival time.

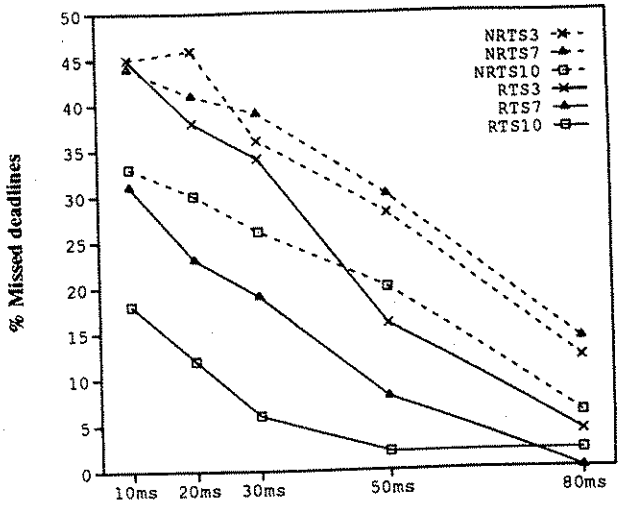

Figure 2: SITES=10, Interarrival time.

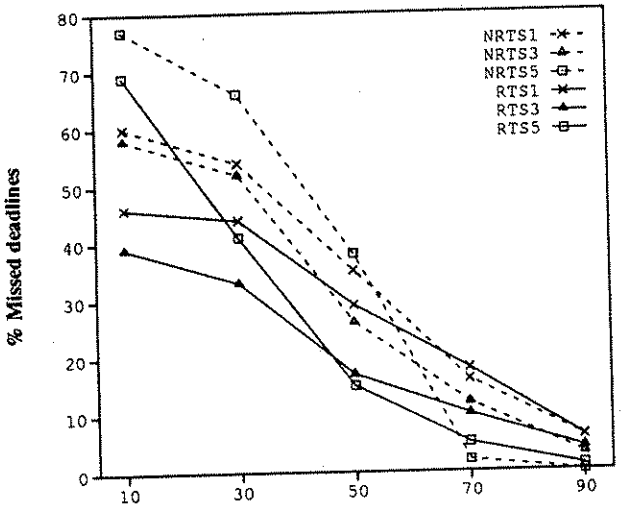

Figure 3: SITES=5, Read only trx %.

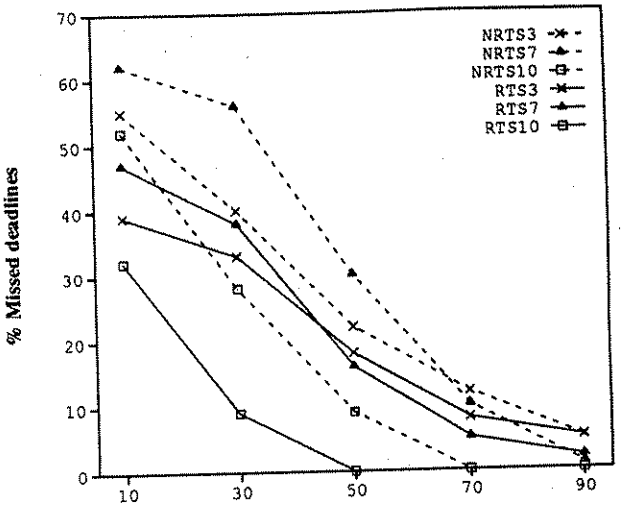

Figure 4: SITES=10, Read only trx %.

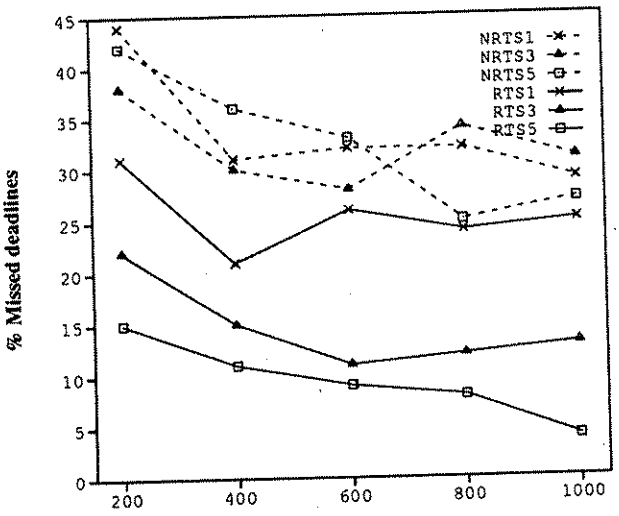

Figure 5: SITES=5, Database size.

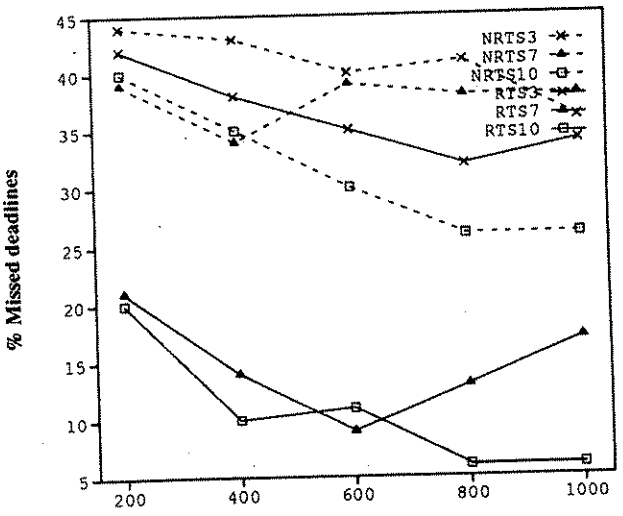

Figure 6: SITES=10, Database size.

conflict with each other and need to be blocked and, as the graphs show, the vast majority of the transactions meet their deadlines.

RTS performs better than NRTS in all cases. When we have a small number of token-sites (i.e. 1 token-site out of 5 sites, or 3 token-sites out of 10 sites), the RTS and the NRTS curves are relatively close to each other. The greater the number of token-sites becomes, the further the respective RTS and NRTS curves veer away from each other. Finally, when all sites are token-sites for each data object, we observe the maximum distance between the curves for the "*% missed deadlines*" of the two schemes.

The experimental results, shown in figures 1 and 2, clearly favor RTS. This was expected, since in the majority of the cases RTS schedules the most urgent transactions first, while NRTS is not at all sensitive to how close a transaction is to missing its deadline. The difference in performance of the two algorithms becomes even greater when the number of token-sites increases. A large number of token-sites means that each update transaction T_u must request consent for commitment from a large number of sites. Therefore, T_u must remain pending for a longer period of time, and consequently T_u will be exposed to more conflicts with other transactions. The increased number of conflicts that must be resolved, combined with the fact that conventional NRTS does not incorporate any intelligent real-time conflict resolution policy, makes the inferiority of NRTS even more apparent.

6.3 Read Only Transactions

We vary the percentage of read-only transactions submitted to the system from 10%, where the majority of the transactions are updates, to 90%, in which case the system behaves almost like a static database, where very few values of data objects change. We assume an average inter-arrival time of 30msec, and a database size of 1,000 data objects.

Figures 3 and 4 show that both RTS and NRTS behave similarly. The respective "*% missed deadlines*" curves start from a very high missed-deadline percentage, in the neighborhood of 10% (which is reasonable given that 90% of the transactions are update ones) and descend very steeply as the percentage of read only transactions increases. In the vicinity of 90%, almost all transactions are queries, very few conflicts occur, and the two approaches perform identically with each other.

When the percentage of read only transactions is near 10%, then the conflict rate between update transactions is very high, and RTS clearly exhibits lower miss rates than NRTS. When the percentage of read only transactions is 10%, the whole burden of scheduling is shifted onto the conflicting-updates resolution policy. NRTS allows time-critical updates to be aborted and restarted in order for less critical transactions to proceed, provided that the database remains consistent. On the contrary, RTS adopts the time criticality of a transaction as the first criterion for resolving conflicts with other transactions: less critical transactions are blocked or even aborted whenever necessary in order for transactions closer to their deadlines to proceed freely. An increased number of token-sites makes the difference in performance even greater for the same reasons as mentioned in the previous section.

The smaller the percentage of read-only transactions becomes, the fewer conflicts occur and the less important role the conflict resolution mechanism plays. Therefore, the two approaches behave almost identically near the 90% point.

6.4 Database Size

We assume an average inter-arrival time of 30msec, and 40% read only transactions. We vary the database size from 200 data objects to 1,000 data objects.

Figures 5 and 6 show that RTS performs much better than NRTS. The distance between the curves is great throughout the spectrum of possible database sizes. When the database size is very small, in the vicinity of 200 data objects, the possibility that two transactions might conflict is very high given that the transaction size remains constant at 12 data objects. As a result, more conflicts between update transactions will have to be resolved. More conflicts means that more sophisticated real-time scheduling is necessary to meet the maximum possible number of deadlines. Moreover, a small-size database causes more queries to conflict with pending updates. In such case, ESR criteria come into play: queries in RTS are allowed to overlap freely with a limited number of updates, and neither of them need to be blocked or aborted as the strict 1SR criteria would require in NRTS. Controlled query inconsistency is a feature that works for the benefit of RTS and leads to further decrease of the missed deadlines percentage.

7. Concluding Remarks

In this paper we have presented a synchronization scheme for real-time distributed database systems. The algorithm is based on a *token-based 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.

What real advantages does our scheme offer the user of a distributed environment? The answer to this question is twofold. There are performance benefits (in the real-time sense), and there are benefits resulting from the fact that this approach is independent of certain restrictive parameters.

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. Additionally, a second mechanism further improves performance: updating the different replicas of the same data object is done asynchronously, but in the same order. Thus, logical write operations become disjoint from the corresponding physical write operations, and update transactions are free to proceed to the next step of their execution or even to commit. Internal database consistency is preserved strictly. Data returned by certain queries are allowed to exhibit limited inconsistency, under user control.

Simulation experiments performed using the distributed database prototyping environment support the above theoretical argument for increased performance. We examined the performance of the new real-time algorithm against the conventional replication control approach under various degrees of system load and data distribution. The base parameters used represent a high load scenario. Even though such high load situations might not arise frequently, it is more informative for the researcher to study the relative efficiency of the algorithms when these "hot spots" occur. In all the experiments, RTS achieves a considerably lower percentage of missed deadlines than NRTS. This is especially evident in the extreme cases of many token-sites, very small inter-arrival times, large percentage of update transactions, or small database size. RTS curves are consistently lower than the respective NRTS curves. This means that RTS misses substantially fewer deadlines than NRTS under identical high load scenarios.

The second advantage of our scheme lies in the fact that there is very little information the user has to provide to achieve efficient system operation. No *a priori* knowledge of the kind or the number of the data objects that are included in the read-set or the write-set of a transaction is needed. The only information required is the kind of each submitted transaction (query or update), and the expected average number of objects accessed by each transaction. Moreover, no execution time estimate is required for each submitted transaction. It would be extremely difficult to compute a run-time estimate, especially in the distributed environments for which our scheme is designed.

There is a price to pay for relaxing correctness criteria and meeting more deadlines. Although the user can control the maximum permissible inconsistency of queries, one can never know exactly which one transaction out of the set of all possibly inconsistent queries will return incorrect data. Note that an overlap counter greater than zero does not necessarily mean that the respective query transaction is inconsistent. It simply indicates that certain RW/WR conflicts were passed unresolved, and inconsistency might be present among the data values returned. Finally, even though we do not need to know the exact length of every transaction, we still need to have available the average expected length of transactions that will be processed by the system, in order to compute the overlap upper bound for each of those transactions.

REFERENCES

[Abb88] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation", Proceedings of the 14th VLDB Conference, Los Angeles, California 1988.

[Abb90] R.Abbott, and H.Garcia-Molina, "Scheduling I/O Requests with Deadlines: a Performance Evaluation", IEEE, Real-Time Systems Symposium, Orlando, Florida, December 1990.

[Ber85] A.J.Bernstein, "A Loosely Coupled Distributed System for Reliably Storing Data", IEEE Trans on Software Engineering, Vol. SE-11, No. 5, May 1985.

[Ber81] P.A.Bernstein, and N.Goodman, "Concurrency Control in Distributed Database Systems", Computing Surveys, Vol. 13, No. 2, July 1981.

[Ber84] P.A.Bernstein, and N.Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", ACM Trans on Database Systems, December 1984, pp. 596 - 615.

[Ber87] P.A.Bernstein, V.Hadzilacos, and N.Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Co., 1987.

[Bre82] H.Breitwieser and M.Leszak, "A Distributed Transaction Processing Protocol based on Majority Consensus", Procs of ACM SIGACT-SIGOPS, Symp. on Principles of Distributed Computing, Ottawa Canada, August 1982.

[Car88a] M.J.Carey and M.Livny, "Conflict Detection Trade-offs for Replicated Data", Technical Report, Un. of Wisconsin, Madison, WI, 1988.

[Car88b] M.J.Carey and M.Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution and Replication", Proceedings of the 14th VLDB Conference, Los Angeles, CA, 1988.

[Car89] M.J.Carey, R.Jauhari, and M.Livny, "Priority in DBMS Resource Scheduling", Proceedings of the 15th VLDB Conference, Amsterdam, 1989.

[Esw76] K.P.Eswaran, "The Notion of Consistency and Predicate Locks in a Database System", Comm. ACM, pp. 624 - 633, Nov. 1976.

[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.

[Hua90] J.Huang, J.A.Stankovic, K.Ramamritham, and D.Towsley, "On Using Priority Inheritance In Real-Time Databases", Dep. Computer and Information Science, Un. of Massachusetts, November, 1990.

[Lam78] L.Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Comm. ACM, Vol. 21, No. 7, pp. 558 - 565, July 1978.

[Lin89] K.Lin, "Consistency Issues in Real-Time Database Systems", Proc. 22nd, Hawai International Conference on System Sciences, January 1989.

[Pu91a] C.Pu, and A.Leff, "Replica Control in Distributed Systems: An Asynchronous Approach", Dep. of Computer Science, Columbia University, New York, January 1991.

[Pu91b] C.Pu, and A.Leff, "Epsilon-Serializability", Dep. of Computer Science, Columbia University, New York, January 1991.

[Sha88] L.Sha, R.Rajkumar, and J.Lehoczky, "Concurrency Control for Distributed Real-Time Databases", ACM SIGMOD Record 17, 1, March 1988, pp. 82 - 98.

[Son87a] S.H.Son, "A Synchronization Scheme for Replicated Data in Distributed Information Systems", Proceedings of IEEE Symposium on Office Automation, Gaithersburg, MD, April 1987.

[Son87b] S.H.Son, "Synchronization of Replicated Data in Distributed Systems", Information Systems, Vol. 12, No. 2, pp. 191 - 202, 1987.

[Son90a] S.H.Son, "An Environment for Prototyping Real-Time Distributed Databases", International Conference on Systems Integration, pp. 358 - 367, Morristown, New Jersey, April 1990.

[Son90b] S.H.Son, and J.Lee, "Scheduling Real-Time Transactions in Distributed Database Systems", 7th IEEE Workshop on Real-Time Operating Systems and Software, Charlottesville, Virginia, May 1990, pp. 39 - 43.

[Son90c] S.H.Son, "Real-Time Database Systems: A New Challenge", Data Engineering, vol. 13, no. 4, Special Issue on Future Directions on Database Research, December 1990.

[Son91] S.H.Son, and S.Kouloumbis, "Performance Evaluation of Replication Control Algorithms for Distributed Database Systems", Tech.Rep., Computer Science Dep., University of Virginia, Charlottesville, February 1991.

[Ste81] R.E.Stearns, D.J.Rosenkrantz, "Distributed Database Concurrency Controls Using Before-Values", ACM SIGMOD Conf. Proc. 1981, pp. 74 - 83.

[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.