Performance Evaluation of Replication Control Algorithms for Distributed Database Systems

Sang H. Son
Spiros Kouloumbis

Computer Science Report No. TR-91-11
May 21, 1991

Performance Evaluation of Replication Control Algorithms for Distributed Database Systems

Sang H. Son
Spiros Kouloumbis

Computer Science Department
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Replication is a key factor for improving the availability of data in a distributed system. Replicated data is stored redundantly at multiple sites so that it can be used by the user even when some of the copies are not locally available due to site failures. A major restriction in using replication is that replicated copies must behave like a single copy; i.e. mutual and internal consistency must be preserved. Many concurrency control algorithms have been proposed for use in distributed fully replicated database systems. Two of the major families of such algorithms are the *Quorum Consensus Approach* and the *Special Copy Approach*. In this paper we examine one algorithm from each of these families, using a detailed model of a distributed DBMS. Each of the algorithms we have chosen is representative of the family to which it belongs, and both span a wide range of characteristics in terms of how conflicts are detected and resolved. The critical performance results not only determine the relative efficiency of the two algorithms, but also help the researcher obtain a better insight into the trade-offs inherent to each respective approach.

1. INTRODUCTION

Distributed systems consist of multiple autonomous computer systems (sites) connected via a communication network. Distributed Database Management Systems (DDBMS) fall into the above category. The main reasons that motivated the design and implementation of such systems are:

–*Increased availability of data.* Replicated copies of critical data can be stored redundantly at multiple sites;

–*Improved fault tolerance.* Redundantly stored data in multiple sites (possibly with independent failure modes) facilitate system access of data in the presence of failures even though some of the redundant copies might not be available (i.e. site crash);

–*Improved data reliability.* Accidentally destroyed copies are constructed from other copies;

–*Enhanced performance.* Queries initiated at sites where the data are stored are processed locally without incurring any communication delays; and

–*Distributed workload.* Subtasks of a query can be processed concurrently in several sites.

Replication of data in a DDBMS is not a panacea, because several new restrictions must be imposed on the replicated database management procedures [10]. Such restrictions are very likely to result in additional cost and increased complexity of the DB manager. Problems related to replication control are:

–*Mutual consistency* of replicated copies must be maintained. All copies of a data object must converge to the same value, and should be identical if all update activities cease;

–*Internal consistency* of replicated copies must be maintained. Invariant relations among data objects within the database must be preserved;

–*Concurrency control* in a distributed system is more complicated than in a centralized system, because information used to make scheduling decisions is itself distributed, and it must be managed properly to make correct decisions;

–*Inherent communication delays* between sites make it impossible to ensure that all copies remain identical at all times when update requests are processed; and

–*Deadlock detection/prevention and recovery* are more complicated than in systems with centralized control.

There are three major families of concurrency control algorithms for replicated data in distributed systems, but before describing them we first present the distributed database model upon which all the algorithms presented in this paper are based.

The main dynamic component of our database model is the *transaction* [5], which is defined to be the atomic execution of a program that accesses shared data. We have two types of transactions in a distributed system:

– *Local transactions*, that access data only in a single site, and

– *Global transactions*, that access data in several sites – communication among sites is required.

In order for transactions to be managed properly and for the results of their execution to be applied consistently on the database, a special process called *transaction manager* runs at each site. Transaction managers supervise interactions between users and the system and forward the transaction operations to the respective *schedulers*.

A scheduler is a process, that performs concurrency control at each site. The replication algorithms presented and evaluated below will be implemented in this scheduler. *Data managers* are lower level processes, running one at each site, that manage the local database. Finally, given the distributed nature and the increased communication burden of such a database system, special processes, called *message servers*, – running one at each site – are needed to take care of the communication protocols between their site and all the others.

The smallest unit of data accessible to the user is the *data object*. A data object is an abstraction that does not correspond directly to a real database item. In a particular system, a data object might be a file, a page or a record. *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. Read and Write operations are used to access data objects, and then local computations are used to determine the value of a data object for a write operation. Such local computations are not of concern for the concurrency and replication control algorithms presented in this paper. The *read-set* of a transaction T is defined to be the set of data objects that T reads, and similarly the *write-set* is the set of data objects the transaction T writes.

Users of a distributed system expect the system to behave as if it executes transactions one at a time to a *logical one-copy database*, even though the actual database system executes many transactions at a time using several replicated data objects. Providing the user with a view of a single-copy data object is a primary concern – and property – of the algorithms to be tested.

We examine two concurrency control algorithms in this study, including one majority consensus based algorithm and one token based algorithm. Each of these algorithms is quite representative of the family that it belongs to, and both present different approaches to conflict detection and resolution. The critical performance results not only determine the relative efficiency of the two algorithms, but also might help the researcher obtain a better insight into the trade-offs inherent in each approach.

The paper is organized as follows. *Section 2* presents the three major families of concurrency control algorithms for distributed databases and gives a more detailed description of the majority consensus approach and the token-based approach algorithms that will be evaluated later on. *Section 3* describes the *Prototyping Environment*, a highly modular simulation environment written in Modula-2, used for the relative performance evaluation of the two algorithms. *Section 4* presents the parameter settings and the invariants of the simulation environment as well as the metrics obtained from the simulation runs. These metrics will be the decisive factors for assessing the efficiency of each algorithm. Performance experiments are described, and the associated results – in the form of graphs – are presented and justified, taking into account the key features of each algorithm. *Section 5* summarizes the advantages and disadvantages of each of the algorithms, discusses the overall simulation policy followed in the experiments and describes possibly improved variations of these algorithms.

2. DISTRIBUTED CONCURRENCY CONTROL ALGORITHMS

Before we get into the formal description of each of the two algorithms in question, we will present some background on the design philosophy for each. The three major families of concurrency control algorithms for replicated data in distributed databases [5] [8] [16] are:

- *Quorum Consensus Approach,*
- *Special Copy Approach, and*
- *Read one copy – Write all copies approach.*

The database in each category can be either fully or partially replicated.

In the *Quorum Consensus Approach*, a voting based scheme is adopted [4]. Each copy has an equal number or different number of votes, and a predetermined number of votes is necessary to perform a desired operation. Queries can be processed locally, while update transactions cannot commit unless a majority of sites consents. Mutual consistency of a majority of copies for each data object is guaranteed; since there must be at least one site that is a member of two different majorities, no two conflicting user requests can be executed and committed. Nevertheless, mutual consistency of the whole system is not strictly imposed, since copies not in the majority set may be obsolete.

There are several approaches that fall into this family. A *simple voting* scheme decides that operations on replicated data objects can be executed only when a transaction obtains necessary locks on a set of copies containing at least a majority. In a *weighted voting* scheme each copy is assigned a fixed number of votes that can be used in gathering a quorum in order to execute an operation on the data. Other schemes, like the *set-based* approach [2] [11], the *loosely coupled* approach [3], the *log-based* approach [12] or the *version-based* approach [6] have been developed based on the quorum consensus approach principle.

All of the above schemes are characterized by a high level of robustness – which is a major advantage of the quorum consensus approach in general – since only a majority of sites need to be operational for transactions with update operations to complete. Much communication overhead is incorporated in those algorithms, but it can be balanced by the fact that they are deadlock-free and can ensure an acceptable response time. We evaluate the *Thomas Majority Approach for Replicated Databases* [18], a member of the Quorum Consensus family.

In the *Special Copy Approach*, the availability of a special copy – or any copy in a set of special copies – enables an operation on a data object. Two major trends occur within this family of algorithms:

- *Primary copy methods*, and
- *Token-based methods.*

The primary copy methods are fairly representative of the philosophy behind this family of algorithms. Each data object is associated with a primary site (or master site) to which all updates for that data object are first directed. Different data objects may have different primary sites. Updates are executed only if the primary copy of the data object is available. The main advantage of the method is its simplicity, while its main drawback is its vulnerability to failures of primary copy sites. Descendents of the primary copy method are the *primary copy scheme with backups* and the *available copy* schemes.

Token-based methods lie between the quorum consensus approach and the primary copy approach. Here, the notion of the *token-copy* is introduced: a token-copy designates a read-write copy which contains the latest version of the data object. Each data object is assigned a predetermined number of tokens. The existence of multiple token-copies for each data object provides the user with a higher availability of data objects than in the primary copy methods, because a data object can be accessed and updated even if some token-sites are unavailable. Token-based methods are different from the quorum consensus approach in the sense that they do not require a quorum in order to operate on a data object. However, they are different from the primary copy methods as well, in the sense that the unavailability of a single token-copy does not necessarily make a data object unavailable. The *true-copy token approach* and the *token-based replication control approach* follow this second trend.

The Special Copy Approach – in an overall evaluation of both of its trends – has low message traffic and improved response time but suffers from reliability problems, especially when there is only one primary copy per object or only a few token-sites. The *token-based replication control approach* [15] is the second algorithm we evaluate.

In the *Read one copy – Write all copies Approach* the name of the family is quite indicative for the policy used by these algorithms. Algorithms falling into this approach can be divided in three basic classes [4]:

- *Locking* algorithms, such as the *Distributed Two-Phase Locking (D2PL)*, and the *Wound-Wait (WW)*.
- *Timestamp* algorithms, such as the *Basic Timestamp Ordering (BTO)*.
- *Optimistic* (or *Certification* algorithms), such as the *Distributed Certification (DOPT)*.

The major advantage of this approach is the strict preservation of the mutual consistency of the *whole database system* against the limited mutual consistency of only a majority of sites that is guaranteed by the majority consensus approaches, or the limited mutual consistency of only the token-sites that is guaranteed by the token-based approaches. However, mutual consistency is sometimes achieved at the cost of increased response time, and algorithms that belong to this last family are more susceptible to deadlocks.

2.1 A Majority Consensus Approach (MCA)

The database model adopted by this algorithm makes the following assumptions:

- Full replication of the database at all sites is assumed;
- An *Application Process (AP)* to which queries and update requests are submitted, is running at each site;
- The *Database Management Process (DBMP)* coordinates the voting procedures and accesses data objects;
- Each AP is associated with one DBMP, and one DBMP is assigned to each site;
- The *base-set* of a transaction is the set of all data objects included in the transaction;
- The *update-set* of a transaction includes only the data objects to be updated;
- Two transactions *conflict* if $\text{Base-set}_1 \cap \text{Update-set}_2 \neq \emptyset$;
- The timestamp that is assigned to a transaction when it is submitted to a site consists of a pair *(t,i)*, where t is the value of the local clock, and i is the unique identifier of the site;

- The timestamp that is assigned to every updated data object in site i is (T_s, i), where $T_s = 1 + \max\{\text{time}, \max\{T_b\}\}$, and T_b is the set of all the timestamps of the data objects in the base-set of transaction T.

Two types of transactions are processed in the Majority Consensus Approach; *query* transactions, that are read only transactions, and *update* transactions, where data objects are both read from and written into the database. A different database manipulation mechanism is adopted for each of the above operations. A detailed flowchart of the majority consensus algorithm can be seen in Figure 1. The following describes the implementation of these two transaction types.

Query: Get base variable values from the local site to which the request was submitted.

Update: The following steps are applied by the DBMP for each update transaction submitted to it:

1. **Vote for the currently submitted update request.**
   - Vote *REJECT* if any of the base variables is obsolete in comparison to the respective copies of the local site.
   - Vote *OK* if base variables are current and there is no conflict with any other pending transaction.
   - Vote *PASS* if base variables are current but there exists a conflict with a pending request of higher priority.
   - *Defer* voting if transaction conflicts with a lower priority request.

2. **Resolve the request after voting.**
   - If the vote was *OK* and a majority was achieved, then the request is accepted, go to step 3.
   - If the vote was *REJECT*, then abort the transaction, and go to step 3.
   - If the vote was *PASS* and no majority consensus is possible, then abort the transaction, and go to step 3.
   - *Otherwise*, forward the request to the next available site (daisy chaining), and go to step 1.

3. **Final decision.**
   - If the transaction was *rejected*, unblock all the transactions deferred by the current one, and resubmit the transaction.
   - If the transaction is *accepted*, send update messages to all sites and reject any transaction that was deferred by the current one.

2.2 A Token-based Approach (TBA)

The database model adopted by this algorithm makes the following assumptions:

- Full replication of the database at all sites is assumed.
- The timestamp assigned to a transaction that is submitted to site i consists of the pair *(t, i)*, where t is the value of the local clock [14].
- No two transactions have the same timestamp, and only a finite number of transactions can have a

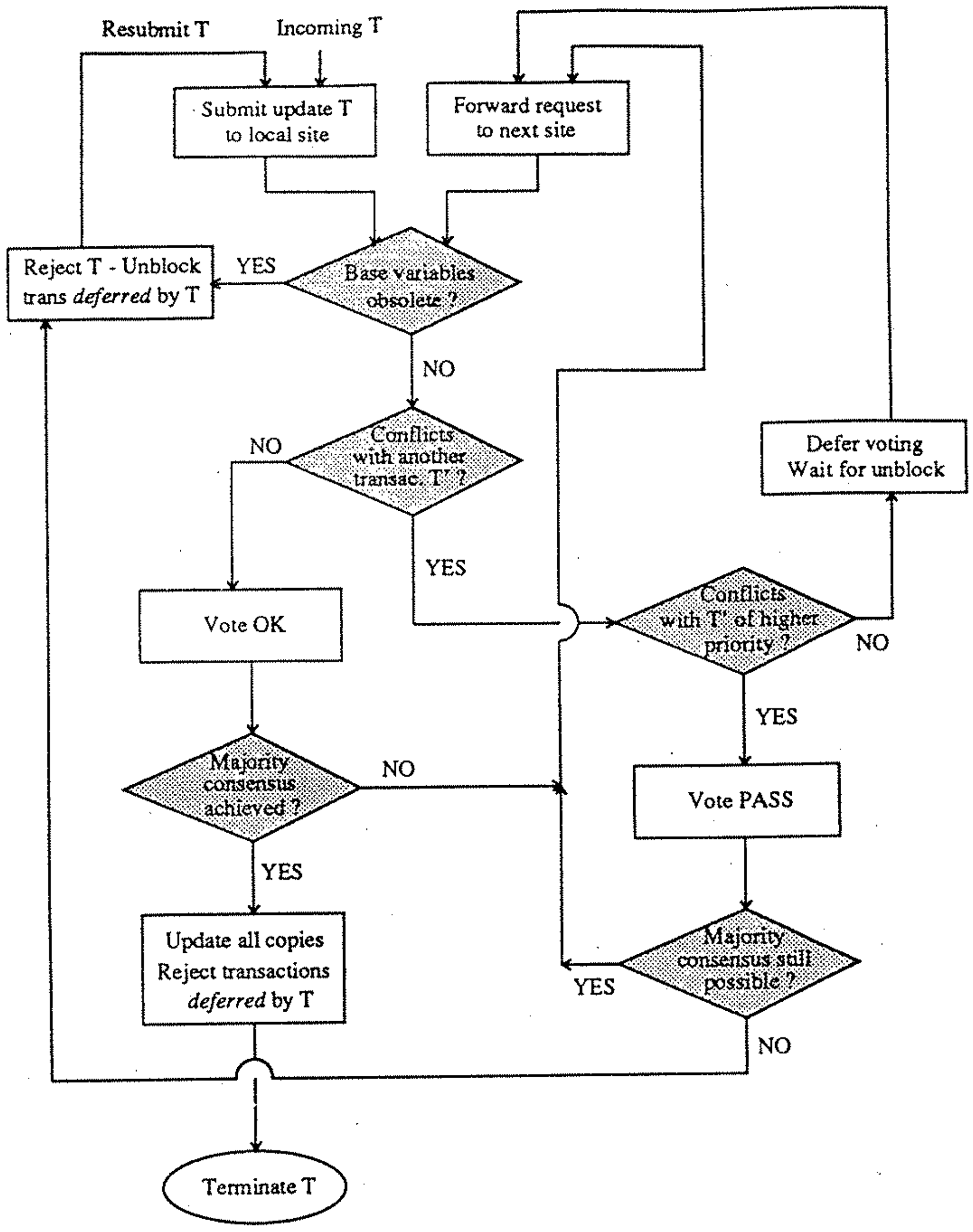
Resubmit T
Incoming T
Submit update T to local site
Forward request to next site
Reject T - Unblock trans *deferred* by T
YES
Base variables obsolete ?
NO
Conflicts with another transac. T' ?
NO
YES
Defer voting Wait for unblock
Vote OK
Conflicts with T' of higher priority ?
NO
YES
Majority consensus achieved ?
NO
Vote PASS
YES
Update all copies Reject transactions *deferred* by T
Majority consensus still possible ?
YES
NO
Terminate T

Figure 1: A Majority Consensus Approach for replicated data.

timestamp less than that of a given transaction.

- Local clock synchronization follows the rules [14]:
  - Each site increments its local clock by one between any two successive events;
  - Messages contain the current clock value of the sender site;
  - On receiving a message with a clock value $t >$ current local clock value, set local clock to $t+1$.
- A transaction T is said to be pending on a read/write request on a data object x whenever a read/write request on x issued by T is accepted and has not yet completed.
- A transaction T stops to be pending on a data object x after the value of x is read or updated accordingly.
- Possible conflicts are:
  - RW: T_1 requests a read operation on x while T_2 has already issued a write request on x;
  - WR: T_1 requests a write operation on x while T_2 has already issued a read request on x;
  - WW: T_1 requests a write operation on x while T_2 has already issued a write request on x.
- Two values are associated with each data object; the *after-value* (new version) and the *before-value* (old version).
- Every transaction T is associated with two lists of transactions:
  - The *before-list (BL)*, a list of transactions that read a before-value of any data object in the write-set of transaction T; and
  - The *after-list (AL)*, a list of transactions which write an after-value for any data object in the read-set of transaction T.

In the token-based approach there is no discrimination between read only transactions (queries) and read/write transactions (updates). All transactions are processed uniformly by executing their read/write operations one by one. However, the procedure for handling read requests on individual data objects is different from the procedure for handling write requests. Figure 2 gives a detailed flowchart of the token-based approach.

- ***Read operation on object x submitted by transaction T.***

```
(r1) If ∃ RW conflict with transaction T' then
       If T' → T ⇒ T waits for T' to terminate;
       If T → T' ⇒ T reads before-value of x,
                    T cannot commit before T' terminates, proceed to (r2);
     end if;

(r2) If r[x] is submitted to a token-site, then
       Value of the local copy of x is returned,
       Read operation r[x] terminates;
     end if;
     If r[x] is submitted to a non-token-site, then
       If timestamp(x) > timestamp(T) then
         Value of the local copy of x is returned,
         Read operation r[x] terminates;
       else
         An Actualization Request Message (ARM) is sent to the next avail-
```

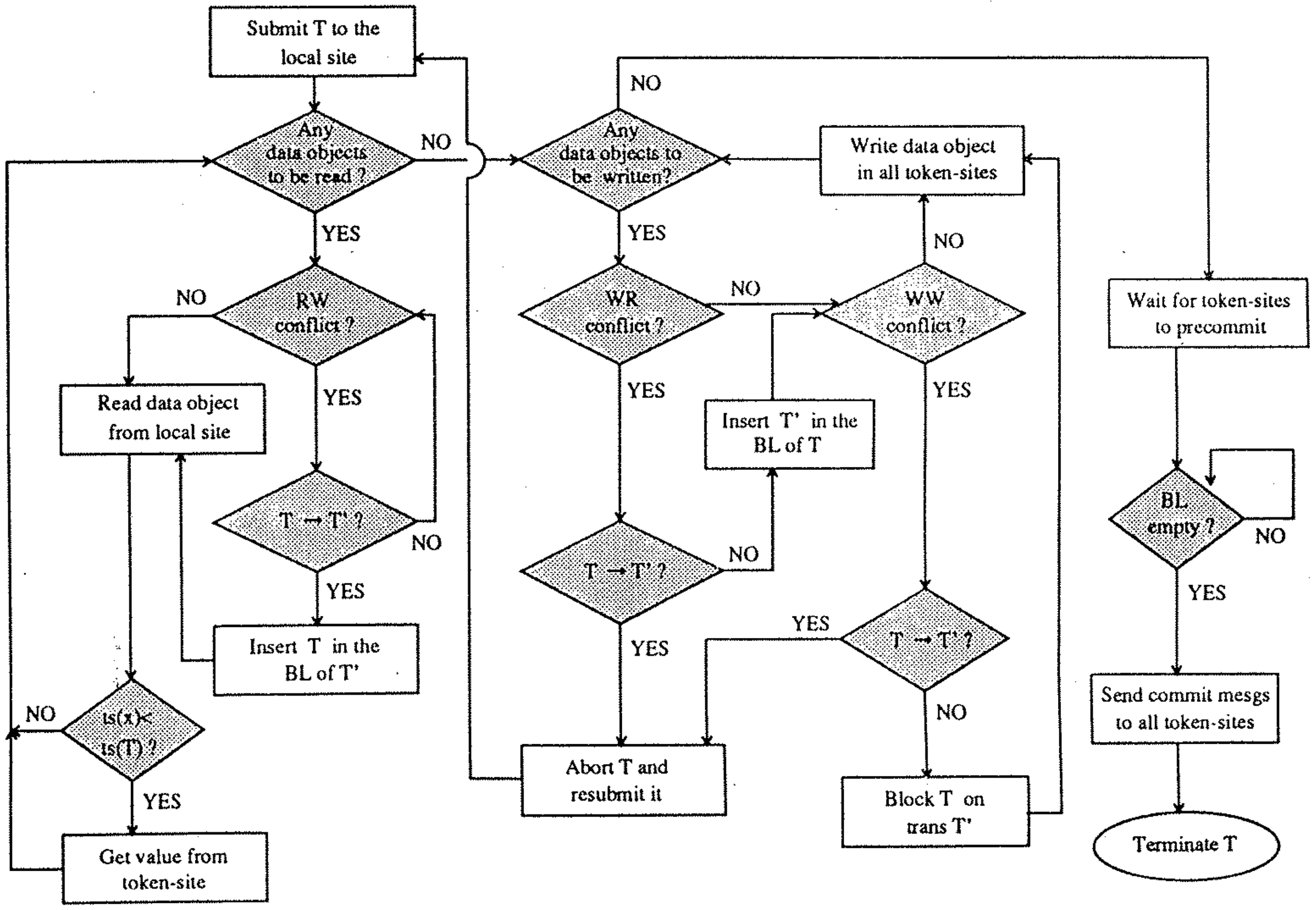
Submit T to the local site
Any data objects to be read ?
NO
YES
RW conflict ?
NO
YES
Read data object from local site
T → T' ?
NO
YES
Insert T in the BL of T'
ts(x)< ts(T) ?
NO
YES
Get value from token-site
Any data objects to be written?
NO
YES
WR conflict ?
NO
YES
T → T' ?
NO
YES
Abort T and resubmit it
Insert T' in the BL of T
Write data object in all token-sites
WW conflict ?
NO
YES
T → T' ?
YES
NO
Block T on trans T'
Wait for token-sites to precommit
BL empty ?
NO
YES
Send commit mesgs to all token-sites
Terminate T

Figure 2: A Token-based Approach for replicated data.

```
        able token-site,
        Local copy of x is updated with the most updated value coming from
        the token-site,
        The value of the token-copy of x is returned,
        Read operation r[x] terminates;
      end if;
    end if;
```

• ***Write operation on object x submitted by transaction T.***

```
(w1) If ∃ WR conflict with transaction T' then
       If  T' → T  ⇒ T 's write request is granted,
                      T cannot commit before T' terminates,
                      Proceed to (w2);
       If  T → T' ⇒ T is rejected;
     end if;

(w2) If ∃ WW conflict with transaction T' then
       If  T' → T ⇒ T is blocked on the termination of T',
                     After T' terminates proceed to (w3);
       If  T → T' ⇒ T is rejected;
     end if;

(w3) Send the write request to all token-sites of data object x and update
     x's value at these sites,
     Updates at non-token-sites may be scheduled after transaction T com-
     mits;
```

• ***A transaction commits when all of the following conditions are true:***

–All token-sites of each data object in the write-set of the transaction have precommitted.

–Each data object in the read-set of the transaction is read.

–There is no active transaction that has seen before-value of any data object in the transactions's write-set.

3. THE DISTRIBUTED DATABASE PROTOTYPING ENVIRONMENT

One of the primary reasons for the difficulty in successfully developing and evaluating new database techniques for distributed systems is that it takes a long time and large effort to develop a system on which to evaluate the techniques. To reduce the time and effort in this approach, the Prototyping Environment [17] has been developed based on a flexible message-passing concurrent programming kernel. The objective of the prototyping environment is to aid the system developers to experiment on new synchronization techniques and to evaluate them. It is organized in a modular fashion to provide enhanced experimentation capabilities. Such division enables the changing or replacing of any of the independent components without affecting the overall structure of the system. Moreover, each component is in turn composed of several modules, the interfaces of which are designed to be general and flexible enough so that software developed using their service calls may be run on actual hardware without major changes.

The prototyping environment is implemented in Modula-2 and runs on Sun workstations. The environment provides the user with multiple threads of execution and guarantees the consistency of concurrently executing processes. Thus, the prototyping environment provides a platform for building and testing our concurrency control schedulers for each of the algorithms in question.

The program is highly concurrent in order to simulate the environment of a real time distributed database system as closely as possible. Several requests can be submitted at the same time, and many read/write operations take place concurrently at different sites. The general structure of the model is presented in Figure 3a. Detailed discussion of the implementation of mechanisms described below falls outside the scope of this paper, therefore the system module descriptions are given abstractly. For the required degree of parallelism to be achieved, the program adopts the following 4 classes of concurrency threads:

3.1 The main program thread

This is the "backbone" of the simulation program. It incorporates the *user interface* [17] that allows the user – either by keyboard input or parameter file – to specify as many system configuration characteristics and load parameters as possible.

System configuration parameters are:

–The *number of sites*;

–The *number of token-sites* when the token-based approach is simulated;

–The *database size*, described by the number of data objects stored at each site;

–The *CPU* and *I/O speed*, in msecs of processing time per operation;

–The *communication overhead*, in msecs per message sent; and

–The *simulation time*, i.e. the number of seconds that the simulation run will last.

Load characteristics that can be specified by the user are:

–The *total number of transactions* that are going to be submitted at the system;

–The *transaction size*, i.e. the total number of data objects that the transaction will access (either read or write);

–The *abort cost*, i.e. the average time in msecs an aborted transaction needs to wait (in resource queues) before it is restarted;

–The *mean inter-arrival time* in msecs between transactions;

–The *percentage of the read only* transactions; and

–The *percentage of objects to be updated* in the total number of data objects that a read/write transaction accesses.

The main program initializes the database system, triggers the execution of the other three threads of concurrency, and at the same time runs the *performance monitor*. The performance monitor interacts with the transaction managers to record the time that each event occurs and prints the statistics for each transaction, such as the arrival, start and end time, the total processing time, the blocked interval, and the number of aborts.

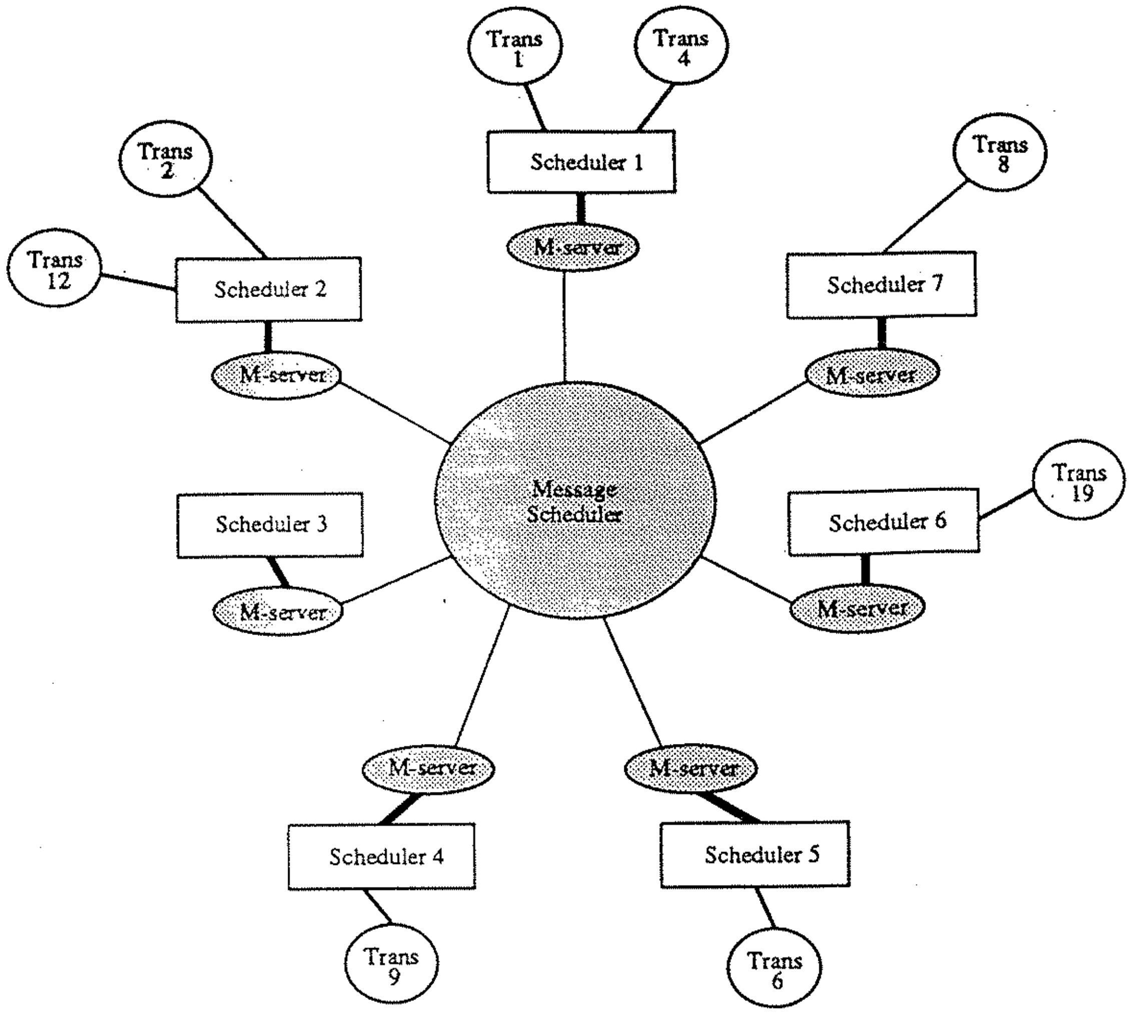

Figure 3a: Prototyping Environment Configuration.

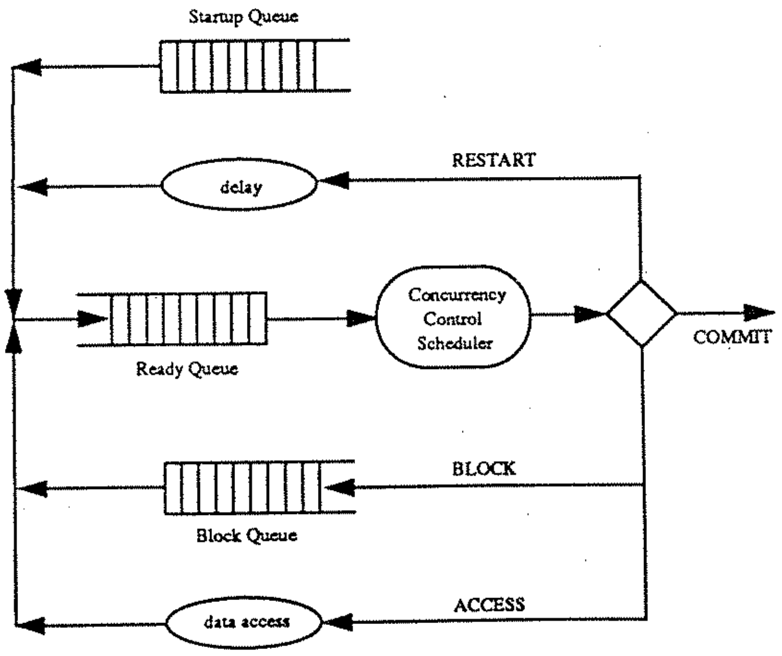

Figure 3b: Transaction Queueing Model.

Finally, the *overall statistics* of the simulation are printed:

–The *total number of transactions completed*, i.e. the transactions that successfully reached completion within the given time limits;

–The *throughput*, or the average number of data objects that were processed per second;

–The *average response time*, the average time – in msecs – transactions spend in the system (i.e. the period starting the moment a transaction is submitted to the system until the moment that all of its reads/write requests are satisfied);

–The *total number of aborts*, indicating how many times transactions were rejected and restarted before they completed; and

–The *percentage of CPU utilization* for each site, i.e. the fraction of the total simulation time during which the CPU of each site was doing some useful computation work for the data objects to be updated.

3.2 Transaction manager threads

One transaction manager process is assigned to each transaction submitted to the system. The manager runs on the site on which the query/update request originated, and interacts with the local scheduler in order to satisfy the individual read/write operations of the request. Figure 3b illustrates the queueing model adopted for transaction processing. A transaction is distinguished from other processes in the system by its behavior – it requests services and receives replies [4].

Only non-periodic transactions arrive at the system. Each arriving transaction is assigned an arrival time, and a read/write set. The read/write set is determined according to the transaction size and the percentage of the data objects to be updated in read/write transactions. The data sets accessed by a transaction are uniformly distributed across the whole database. Globally unique timestamps are generated for each transaction as well. Timestamps consist of two fields: the *clock value* and the *node number*. When a timestamp is to be assigned to an event (e.g., when a transaction arrives), its node's clock value is assigned to the timestamp's clock value. Global uniqueness is guaranteed by the inclusion of the node number field, given that each node is numbered uniquely across the system.

The algorithm followed for each transaction, i.e., whether or not it requests all its locks at once (or asynchronously), or what action it takes if a request for service is denied, is user specified. The user writes his own transaction procedure. At initialization time, this procedure is made into a transaction process according to the user's specification.

3.3 Scheduler threads

One scheduler thread runs at each site. The scheduler captures the semantics of the given concurrency control algorithm and is the only module that must be changed from algorithm to algorithm. The underlying database structure and the data access methods need not be modified. The scheduler is responsible for handling concurrency control requests made by the transaction manager [4], including read and write access requests and several types of other operations (depending on the replication control algorithm that is implemented) as well as requests for permission to commit a transaction. Results of its actions are returned directly to the respective senders.

Each transaction is served in an exclusive manner by the corresponding scheduler. In other words, only one transaction can each time be served by a specific scheduler and all the other pending transactions on the same scheduler will have to wait for their own time slot to access the scheduler.

Currently, we have implemented two replication control schedulers: a *majority consensus scheduler*, and a *token-based scheduler.*

3.4 Message server threads

At each site one message server process controls the message traffic between transactions submitted at this site and the scheduler running at the same site, as well as messages exchanged between the local scheduler and schedulers at other sites. The interprocess communication structure is an independent subsystem, and is designed to provide a simple and flexible interface to the client processes of the application software, independent of the low-level hardware configurations. Application-level processes need not know the details of the destination device. The invariant built into the design of the interprocess communication interface is that the application-level sender allocates the space for a message, and the receiver deallocates it. Thus it is irrelevant whether or not the sender and the receiver share memory space. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocessors with shared memory space. When prototyping the latter, only addresses need to be passed in messages, without intermediate allocation and deallocation.

A *network manager process* coordinates the execution of the message server at each site. The network manager encapsulates the model of the communication network (currently quite simplistic), acting as a switch for routing messages from site to site. Of course, given that the characteristics of the network are isolated in one module, it would be a simple matter to replace the current model with a more complex one that could, for instance, capture the behavior of a wide area network.

4. EXPERIMENTS AND RESULTS

In this section we present our performance experiments for the two replication control algorithms of interest under various assumptions about the transaction load, the transaction inter-arrival time, the communication cost, the percentage of update transactions in the total number of the transactions submitted to the system during the simulation period, and the abort cost per transaction. We first discuss the performance metrics of interest [9] and the parameter settings [1] [9] used, and then interpret the performance results, taking into account the individual features of the two algorithms.

4.1 Metrics and Parameter Settings

The performance metrics employed in this paper are the *throughput* of the system, the *average response time*, and the *number of transactions aborted* during each simulation [8]. The throughput of the system is defined as:

$$\text{throughput} = \frac{(\text{number of transactions completed}) \times (\text{number of data objects accessed per transaction})}{\text{total simulation time}} \text{(rec/sec)}$$

In the above definition, we must emphasize the fact that only records processed by committed transactions are included in the throughput formula. Therefore, the throughput values we get during simulation experiments represent only the useful work done by transactions, and information concerning records accessed by aborted transactions is discarded. In this way, we do not run the risk of being misled by a high throughput in an experiment where the majority of transactions were aborted, and most of the work reflected in the throughput was actually wasted.

The average response time for a transaction is:

$$\text{response time} = \frac{\sum_{i=1}^{n} (t_{\text{end}}^{(i)} - t_{\text{start}}^{(i)})}{n} \text{(sec)}$$

where n is the number of transactions that committed, and $t_{\text{start}}^{(i)}$, $t_{\text{end}}^{(i)}$ are the times that transaction i was submitted to the system and committed, respectively.

Note that a transaction might be aborted and restarted several times before it commits, thus the total number of aborts may be larger than the total number of transactions submitted to the system.

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.

Certain parameters that determine system configuration and transaction characteristics remain fixed throughout the experiments [1] [7]: 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%), and the *abort cost* for each transaction (19 msec). 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 would occur frequently. The high conflict rate allows the concurrency control 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 1.

Parameters used as independent variables in one-variable functions describing the performance metrics mentioned above are the *total number of transactions* submitted to the system (metrics were measured after all of these transactions had completed), the *mean inter-arrival time* of transactions (varying between the two extreme values of 0.5msec and 100msec), the *communication cost* (varying from 1msec to 10msec), and the *percentage of read/write transactions* submitted to the system (varying the number of read only transactions from 10% up to 90%). Some simulation results were recorded using the abort cost (varying from 20ms to 400ms per aborted transaction) as the independent variable of the performance metrics. The values of system configuration and transaction parameters used for simulations are listed in Table 2.

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
Preparation cost/transaction (msec)	1.0 - 5.0
Termination cost/transacion (msec)	1.0 - 5.0
Simulation time (sec)	1000
Transaction size (data objects)	12
Mean inter-arrival time (msec)	30
Read only transactions (%)	40
Data objects to be updated (%)	40

Table 1: Base parameter values.

Parameter	**Values**
Total number of transactions	50, 75, 100, 125, 150
Mean inter-arrival time (msec)	5, 15, 30, 60, 100
Communication cost (msec)	0.5, 1, 4, 7, 10
Read only transactions (%)	10, 30, 50, 70, 90
Abort cost per transaction (msec)	20, 100, 200, 300, 400

Table 2: 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*:

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

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 the simulation time expires.

Five major categories of experiments were performed. Each category covers 3 different degrees of distribution, namely distributed databases consisting of 3, 6 and 10 sites. For each separate case of n sites ($n \in \{3, 6, 10\}$), various token-site numbers are considered. In each category of experiments we vary one of the parameters in Table 2 while keeping the others fixed. Measurements are recorded for each of the three metrics: *throughput*, *response time*, and *number of aborts*. The type and contents of graphs drawn for each metric at a given experiment are summarized in Table 3.

Sites = 3	MCA sites: 3 → line 1 TBA token-sites: 1 → line 2 2 → line 3 3 → line 4	Graph a
Sites = 6	MCA sites: 6 → line 1 TBA token-sites: 1 → line 2 3 → line 3 5 → line 4 6 → line 5	Graph b
Sites = 10	MCA sites: 10 → line 1 TBA token-sites: 1 → line 2 3 → line 3 5 → line 4 8 → line 5 10 → line 6	Graph c

Table 3: Types of graphs.

Due to space considerations we cannot present all our results but have selected the graphs which best illustrate the differences in performance of the algorithms. We have omitted results of experiments where we varied the transaction size, or where we used other degrees of distribution than 3, 6 and 10 sites. The latter experiments confirm the results of the experiments presented in the following sections.

4.2 Experiment 1: Transaction Load

We assume an average inter-arrival time of 30msec, a low communication cost of 1msec, a low abort cost of 19msec and that 40% of incoming transactions are read only. We vary the total number of transactions submitted to the system from 50 to 150.

Figures 4a, b, c show that MCA has a higher throughput than TBA in all cases. When we have a small to medium number of sites (3 or at most 6), then the TBA version that performs best, has the same number of token-sites as system sites. The throughput curves for these TBA configurations of 3 and 6 token-sites lie the closest to those for the MCA. However, when the number of sites grows to 10, TBA configurations of 1 and 3 token-sites are the best, performing the closest to the MCA throughput.

In the cases of 3 and 6 sites, when the number of token-sites is close to the total number of sites, fewer Actualization Request Messages need be sent to token-sites because the possibility that the current site is a token-site is increased, and read requests in TBA can be satisfied immediately. When the number of sites is

10, the 60% of transactions that require update operations becomes the deciding factor. In that case, numerous token-sites cause the update process of TBA to request more communication interactions between sites, while a moderate number of token-sites (3 or 5) balances the high cost of read requests with the low communication overhead of write requests. On the other hand, MCA throughput is essentially insensitive to the number of sites since read requests are always processed locally, and they do not incorporate any communication overhead. Write requests are based on the majority mechanism – the required majority in 6 sites is 4 sites (66% of the database) versus the required majority in 10 sites which is 6 sites (60% of the database) – which explains the slightly better performance of the MCA in the 10 sites configuration.

Figures 5a, b, c show that MCA has a very low response time, which is approximately equal in all cases (3,6,10 sites) to the response time of the TBA configuration where the number of token-sites equals the number of sites. Response time curves of the rest of the TBA configurations (number of token-sites < number of sites) are very close to each other but are much higher than the curve of the MCA. An important factor determining the response time is the availability of data objects at the site where a request is submitted. Database configurations with the maximum number of token-sites guarantee that the access to the local copy will yield the most up-to-date copy of the data object with no additional communication overhead on the transaction's mean response time. Update requests in the TBA do not affect the response time as drastically, since part of the update procedure can be executed after the transaction commits. In the MCA, read only transactions always take the shortest possible time to execute, since local copies are always accessed, whether they contain the most up-to-date value or an outdated value of the data object. In MCA, update operations of a transaction T can execute concurrently for all the data objects in the write-set of T on each of the majority sites, since each of the data objects addresses the very same set of the sites that voted for T. The last two arguments explain sufficiently why MCA exhibits low response time in our experiments.

Figures 6a, b, c show that the number of aborts of MCA is much larger than that of any TBA configuration. Actually, the slopes of all the TBA abort curves are almost flat. This clearly indicates that TBAs remain independent from the transaction load. The slopes of all the respective MCA curves are very steep, which indicates the increased sensitivity of MCA to changes in the transaction load. The above behavior can easily be explained given the fact that TBA adopts a blocking policy for conflict resolution (aborts are avoided unless they are inevitable), while MCA aborts transactions in the majority of the cases where a conflict occurs.

4.3 Experiment 2: Transaction Inter-arrival Time

In this experiment we vary the inter-arrival time of transactions from 0.5msec (for a heavily loaded system), to 100msec (for a non-saturated system state). A low communication cost of 1msec and a low abort cost of 19msec are assumed. Forty percent of the transactions submitted to the system are read only.

Figures 7a, b, c show that MCA has higher throughput than TBA. MCA's throughput curves originate from a local minimum at 5msec and peak at about 60msec. Then the curves descend to a new minimum throughput value at 100msec. When the inter-arrival time is 5msec the system is overloaded by a huge number of transactions striving to access a limited number of data objects (hot spot). Consequently, the number of conflicts becomes so large that the majority of transactions is aborted. Recall that MCA resolves conflicts by aborting transactions and restarting them. In this case, although a large number of records are processed, no real contribution is made to the system throughput as long as many of those records belong to transactions that are aborted. As the inter-arrival time increases, the number of active transactions per second in the system decreases. Fewer transactions attempt to achieve consensus from the majority of the sites, fewer conflicts occur and more transactions can be served properly without being aborted. When the inter-arrival time is raised above 60msec, the number of active transactions per second becomes less than the actual number of transactions that can be served by the system per time unit. The observed degree of throughput is the natural result of an underutilized system.

As in experiment 1, TBA consistently achieves a lower throughput than MCA, for similar reasons. TBA performs best in the case of 3 and 6 system sites when there are 3 and 6 token-sites respectively, and the system configurations with fewer token-sites (1 or 3 token-sites) perform best when the system consists of 10 total sites. The throughput curves are very flat in the majority of cases. Thus, TBA's throughput is independent from the system load. Given that TBA adopts a blocking policy [4] and is deadlock free, there is an upper bound to the number of transactions that can be served efficiently. Once the arrival rate is high enough that the system becomes saturated the lower priority transactions are blocked on waiting queues until higher priority transactions terminate. A few of these low priority transactions may be aborted, but not enough for the system throughput to degrade. Assuming that the system is free from data contention – a reasonable assumption since there are no limits in the size of blocking queues – younger transactions will always be blocked behind older ones while the system maintains a constant throughput rate by processing transactions in a FCFS fashion. However, signs of underutilization are apparent when we have 10 sites and transactions arrive every 100msec.

Figures 8a, b, c illustrate the same relationship between the average response times of the two algorithms as seen in Figures 5a, b, c. The higher the arrival rate is the longer the response time for both approaches. MCA has a long response time due to the fact that a heavily loaded system might cause a transaction to be aborted several times before it commits. TBA's long average response time is due to the long period during which transactions are blocked in queues waiting for higher priority transactions to terminate. Still, MCA has the lowest average response time, since abort cost is relatively low (there is no resource contention), and thus it is preferable to abort a conflicting transaction and to restart it than to block it on a long waiting queue.

When the inter-arrival time increases the number of conflicts is reduced, and transactions spend less time in blocking queues or being restarted. As a result, average response time is reduced.

4.4 Experiment 3: Communication Cost Considerations

We assume an average transaction inter-arrival time of 30msec, an abort cost of 19msec and that read only transactions comprise 40% of the total transactions submitted to the system. We vary the communication cost between sites from 0.5msec to 10msec.

Figures 9a, b, c show that MCA has higher throughput than TBA. When the number of sites is 3 or 6, TBA configurations with all sites being token-sites perform best among all the TBA versions, and the performance curves are closest to the respective MCA curves. When the number of sites is 10, the best performing TBA configurations are the ones with 1 and 3 token-sites. Having only 3 or at most 6 sites introduces low communication needs among sites, and traffic is constrained by the low number of nodes in the network. The more token-sites there are, the higher the throughput that can be achieved. By increasing the number of sites to 10, the communication traffic increases dramatically with the number of token-sites and diminishes any performance advantages gained by any relatively large number of token-sites (discussed in experiment 1).

Now consider the distance between the MCA throughput curve and the TBA curve of the best performing TBA configuration: notice that the distance increases as we move from the left to the right on the communication cost axis. To understand this relative throughput ordering between MCA and the best TBA version, we must consider the update mechanism for each of the two approaches. TBA guarantees that at least a certain number of token-sites for each data object must store the most recent value of that object. Hence, a minimum set of sites will always be consistent for each data object, whereas MCA guarantees that the majority of system sites will consistently store the most recent value of each data object. At first glance, it seems that TBA should perform the same as MCA does when the number of token-sites is more than the 50% of the total number of sites. However, this is not the case: MCA ensures that new values of all data objects in the write-set of a transaction T are installed consistently in the same set of sites for each data object, while TBA (assuming token-sites > 50% of system sites) ensures exactly the same but for a different set of token-sites for each data object in the write-set.

For a majority of m sites that voted for an update transaction T (cardinality of T's write-set = k) in a distributed system of n sites (m > n/2), MCA needs to send at least m messages, one to each of the sites of the majority, whereas TBA (with m token-sites) needs to send m·k messages. Messages sent by MCA to sites out of the majority set or sent by TBA to non-token-sites are not taken into account, since this can be done after the transaction commits and there is no direct impact on system throughput. TBA introduces much more communication overhead among sites. This is evident in the relative difference of the throughput curves in all of the Figures 9a, b, c as the communication cost rises from 0.5msec to 10msec.

Figures 10a, b, c which illustrate the average response time of the two approaches, support the previous analysis. In addition to the number of messages, the speed with which read requests are served greatly influences the difference between average response time curves of MCA and TBA. MCA achieves the maximum speed by always accessing local copies, whereas TBA might frequently need to go through the network by sending Actualization Request Messages to token-sites and receiving the requested values. This causes TBA to perform even more poorly under high communication costs, especially when the percentage of token-sites is very low.

4.5 Experiment 4: 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, an abort cost of 19msec and a low communication cost of 1msec.

Figures 11a, b, c show that both MCA and TBA behave similarly. The respective throughput curves start from a very low throughput level in the neighborhood of 10% – which is reasonable given that the 90%

of transactions are update ones – and climb very steeply as the percentage of read only transactions increases. In the vicinity of 90%, MCA and all versions of TBA reach their maximum performance. The TBA versions that perform best are the ones with all system sites acting as token-sites. Although these TBA versions handle read requests identically to MCA – local copies of data objects are immediately accessed – TBA throughput curves are still lower than the respective MCA curves. This downward shifting is due exclusively to update requests. For a system consisting by n sites, TBA requires update messages to be sent to and acknowledged by n sites (token-sites = sites), whereas MCA requires update messages to be sent to (n/2 + 1) sites, thus MCA is slightly superior.

As far as the relative behavior of the various TBA configurations is concerned, we see that the more token-sites we have, the better the respective TBA version performs. The small throughput difference between TBA versions at 10% read only transactions is due to the slightly increased communication overhead in versions with fewer token-sites having heavier traffic of Actualization Request Messages (ARM). The increased percentage of read only transactions makes TBA's throughput more sensitive to the number of token-sites, since the demand for ARM's in system configurations with fewer token-sites becomes more and more severe. Experimental results verify the theoretical assessment: the higher the percentage of the read only transactions becomes, the further apart from each other TBA curves fall.

The data in Figures 12a, b, c emphasize the fact that TBA with all sites as token-sites performs approximately the same as the MCA. Actually, in the case of 3 sites, TBA achieves an even better average response time than that of MCA. Both the curves for MCA and TBA start with a long response time, each of them for a different reason.

MCA exhibits high response time at the neighborhood of 10% due to the fact that 90% of the transactions are update transactions, and consequently large number of conflicts occur. This is evident in Figures 13a, b, c. In these figures, MCA exhibits a very high abort rate, which accounts for the high response time observed in Figures 12a, b, c.

For TBA the situation is a little different. The abort curves illustrated in Figures 13a, b, c are almost flat – they are comparable to the abort curves in Figures 6a, b, c presented in experiment 1. This means that TBA's abort rate is insensitive to the transaction type that dominates the total system load. Nevertheless, average response time of the TBA approaches is still high at 10% read only transactions. The reason for this can be found in the way TBA resolves conflicts. Given that TBA adopts a blocking policy, long queues of blocked transactions result in a long response time. Of course, once the percentage of read only transactions becomes larger, conflicts in both approaches exponentially decrease, and the same thing happens to the respective average response times. In the cases of 6 and 10 sites, MCA is a little ahead of TBA for the same reasons explained in the throughput analysis above.

4.6 Experiment 5: Abort Cost Considerations

We assume an average inter-arrival time of 30msec, a low communication cost of 1msec and 40% read only transactions. We vary the abort cost from 20msec (no resource contention) to 400msec (high resource contention).

Figures 14a, b, c show that MCA is strongly affected by the abort cost, while TBA remains virtually insensitive to it. MCA's throughput curve starts from a high throughput value and descends very abruptly as abort cost increases. TBA's curves, for any number of token-sites, remain almost horizontal. The relative order of the TBA curves is the same as in all previous experiments for the same reasons explained before. When the abort cost exceeds the 100msec value, MCA's throughput falls under the best performing TBA version (token-sites = system sites), while an abort cost over 200msec causes MCA to perform worse than any TBA version.

The deciding factor in the behavior of the two algorithms is the number of transactions aborted during execution. In MCA transactions conflicting with higher priority transactions are aborted in the majority of the cases. A high abort cost combined with a large number of abort actions results in a resource contention situation: the majority of transactions are blocked on resource queues waiting for resources to become available. Very little progress can be made in such cases, and an extremely low throughput is the natural consequence.

With TBA, transactions are blocked in queues when they conflict with higher priority transactions, and they remain blocked until the latter terminate. Very few transactions are aborted, and thus very few need to be restarted. A high abort/restart cost would only affect this small percentage of aborted transactions, therefore TBA curves are relatively flat even when the abort cost becomes very high (i.e. 400msec).

The average response time curves in Figures 15a, b, c, tend to reflect an analogous behavior to that in Figures 14a, b, c. Response times of the TBA versions are insensitive to variations of abort cost, while MCA response time curves climb steeply as the abort cost increases. Again, the TBA version with all sites as token-sites performs the best in the majority of the cases with the lowest average response time. MCA has a much worse response time, especially at higher values of the abort cost. TBA versions with the number of token-sites less than the total number of sites behave somewhere in between.

5. CONCLUSIONS

In this paper we have studied two algorithms that belong to the two most representative families of replication control algorithms for distributed databases, the *Majority Consensus Approach (MCA)* and the *Token-based Approach (TBA)*. We examined the performance of these algorithms under various degrees of system load, message cost, data distribution, and abort cost. Our simulation results illustrate the performance tradeoffs between the two algorithms.

First, we observe that our base parameters represent a high load scenario [1] (relatively high number of conflicts). Even though such a high load may not arise frequently, a system should maintain the highest possible throughput combined with the lowest feasible average response time when these "hot spots" occur.

A second observation concerns the subjectiveness introduced when we transformed the abstract specifications and logical flow of the two algorithms into running simulation programs. Additional effort was put into the design and implementation of the schedulers corresponding to the two algorithms in question in order to eliminate the subjectiveness factor. The primary goal was to implement the conflict resolution mechanisms as closely to the original description as possible. A second goal was to place time delays (i.e. due to communication costs, I/O, CPU processing) at the right point in the simulation programs, so that the behavior of the system would be identical to an equivalent real world application.

The *Majority Consensus Approach* guarantees a high throughput and a good average response time, provided that there exists no resource contention. However, when resource contention appears – i.e. abort/ restart costs grow larger – MCA performs poorly due to the huge number of transaction aborts triggered by the scheduler. Moreover, the plethora of abort actions cause a lot of work to be wasted, since numerous transactions are rejected in the middle of their execution.

Furthermore, consistency of queries cannot always be guaranteed. A read request over a data object x is satisfied immediately by reading the value of x in the local site where the transaction was submitted. Even though a low response time is thereby guaranteed for read requests it is possible that an out-of-date value of the data object x will be read. This happens when the local site is not in the majority set of the transaction that last updates x. Finally, MCA exhibits an increased sensitivity to system load. In situations where the system is heavily loaded, MCA's throughput seriously degrades.

A *Token-based Approach*, on the other hand, guarantees the consistency of read operations in a strict manner. The value returned as the result of a read operation on a data object x is always the most updated one. If the read operation is submitted to a token-site for x, then the value returned will be up-to-date by the very definition of a token-site. If the read operation is submitted to a non-token-site for x, then if the local value is obsolete, an ARM will be sent to the next available token-site for x in order to get the current value. This latter case introduces much communication overhead and makes the algorithm perform poorly.

Storage of data objects is distributed homogeneously among sites. When a transaction T updates data objects, the scheduler ensures that each one of them is written at least in its token-sites. The token-sites are most likely different for each data object in the write-set of T. In case of a failure of a number of sites, it is very unlikely that the updated values of all data objects written by T are going to be lost, because updated values for each data object address a different set of token-sites. On the contrary, in MCA, all updates for data objects in T's write-set address the very same set of sites, the majority consensus set, and if this set of sites fails, then all updated values coming out of T are going to be lost.

Since TBA adopts a blocking policy, it exhibits a very low abort rate, which means that wasted work due to abort actions is minimized. Therefore, TBA's throughput is stable and independent from the current system workload, provided that data contention problems do not arise. The only weak point of TBA is the lower throughput and the higher response time relative to MCA.

At this point, a natural question arises: which of the two algorithms offers the best solution for the replication control problem in distributed databases? It is difficult to make an absolute statement, because system configuration and user requirements of a specific application determine the relative efficiency of the two approaches. In general, MCA is preferable when resource contention is low, high throughput/low response time are required and consistency constraints are not strictly imposed on read operations. However, when resource contention is high and the user does not make any compromises on the up-to-date validity of data returned by queries, TBA offers the only acceptable solution.

In terms of future work, it would be interesting to evaluate other algorithms for replication control in distributed databases, using the Prototyping Environment. The *Symmetric Multiple Copy Update Algorithm* [13] appears to be an interesting approach – it falls into the Quorum Consensus Approach – and it is expected to perform even better than MCA, given that only a consensus of a number of sites on the order of $\sqrt{N}$ is requested. *Distributed Optimistic Two-Phase Locking*, on the other hand, would demonstrate the impact of using a strict and rather conservative mechanism for ensuring consistency, on response time and transaction throughput. Majority Consensus Approach could be extended into the general *Quorum Consensus Approach* [4] to guarantee mutual consistency of queries. MCA and TBA would then ensure the same degree of database consistency for all types of operations, and a new series of experiments could be performed under equivalent constraints.

Site failures need also to be considered in the performance evaluation of all of the above algorithms, as well as for algorithms specifically designed to tolerate them. The Prototyping Environment can then be used to shed light on fault tolerance issues embedded in the existing replication control mechanisms.

Throughput = F(total number of transactions submitted to the system)
Interarrival time: 30ms
Communication cost: 1ms
Read only transactions: 40%
Transaction size: 12 data objects

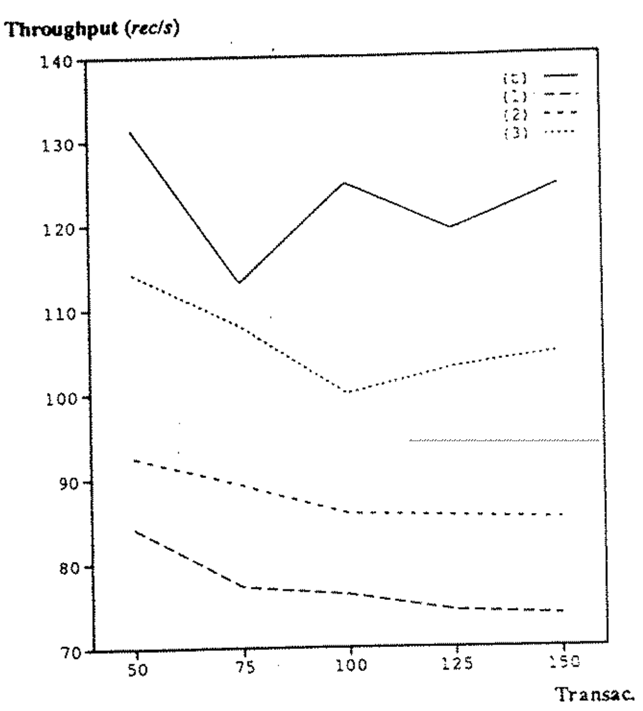

Figure 4a: SITES = 3

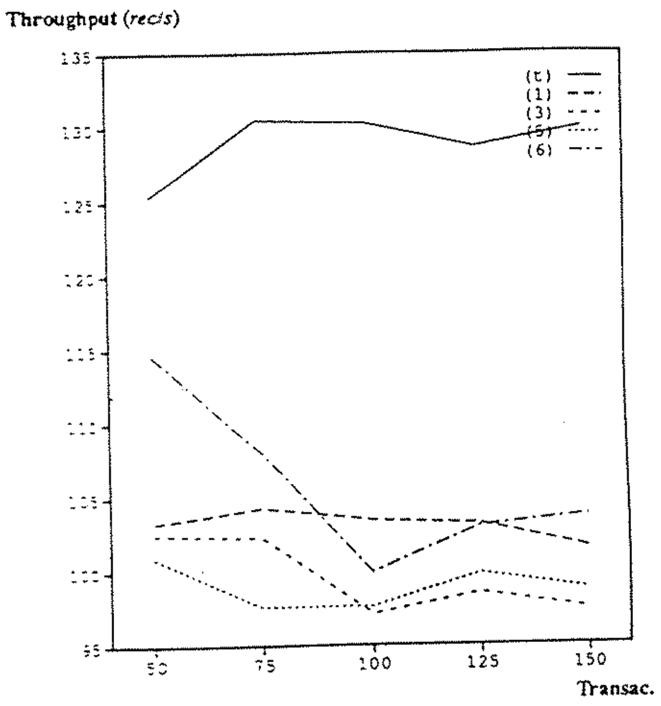

Figure 4b: SITES = 6

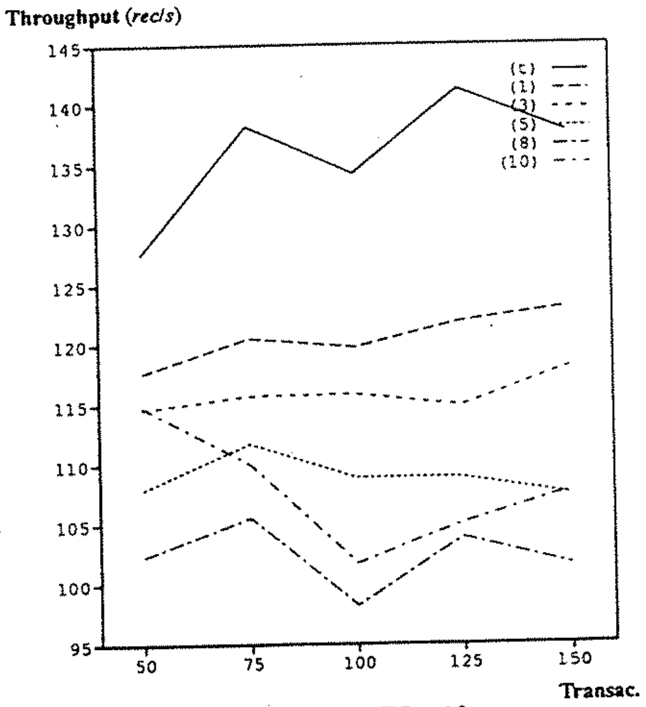

Figure 4c: SITES = 10

Response time = F(total number of transactions submitted to the system)
Interarrival time: 30ms
Communication cost: 1ms
Read only transactions: 40%
Transaction size: 12 data objects

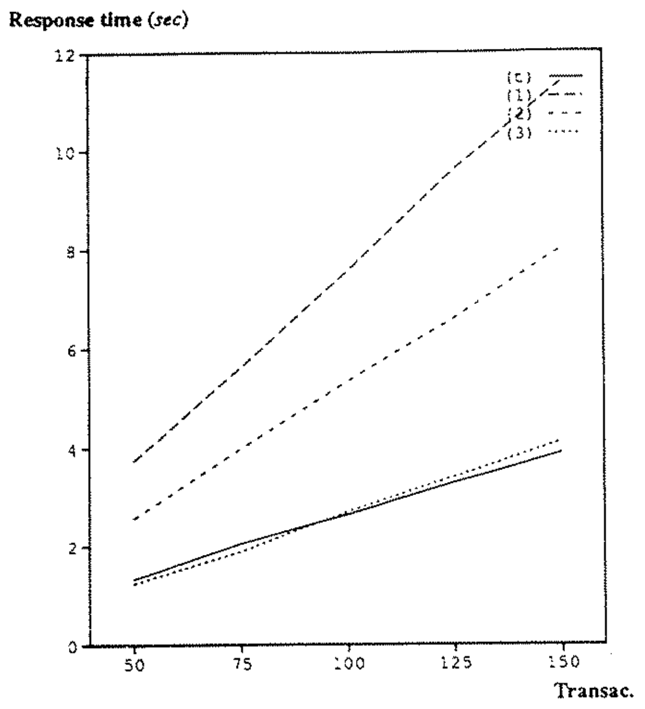

Figure 5a: SITES = 3

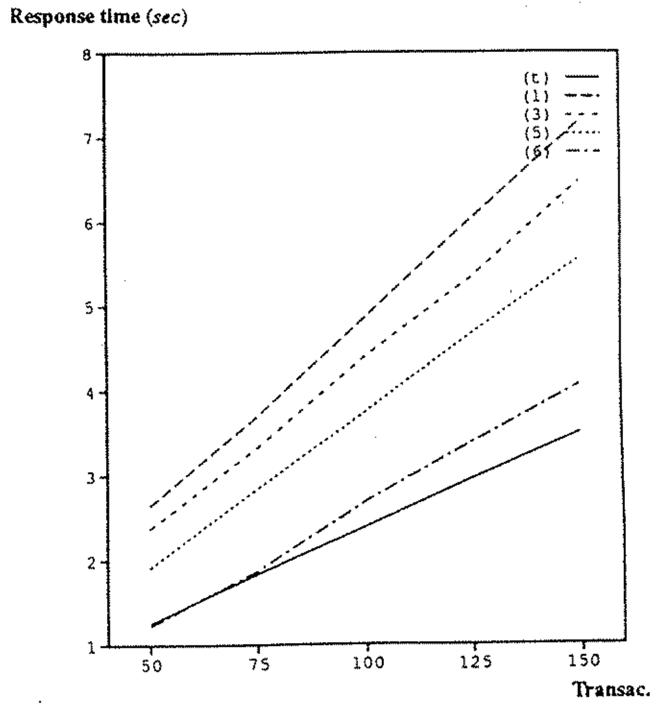

Figure 5b: SITES = 6

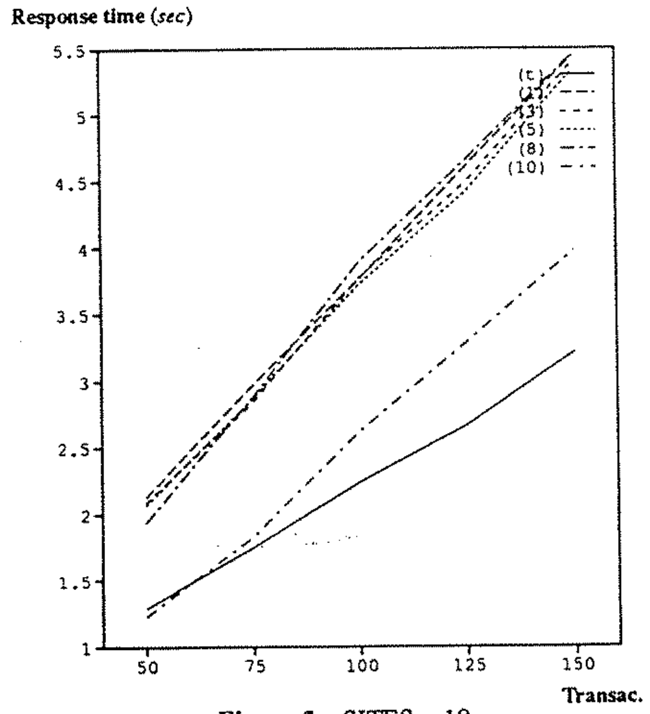

Figure 5c: SITES = 10

Total number of transaction aborts = F(total number of transactions submitted to the system)
Interarrival time: 30ms
Communication cost: 1ms
Read only transactions: 40%
Transaction size: 12 data objects

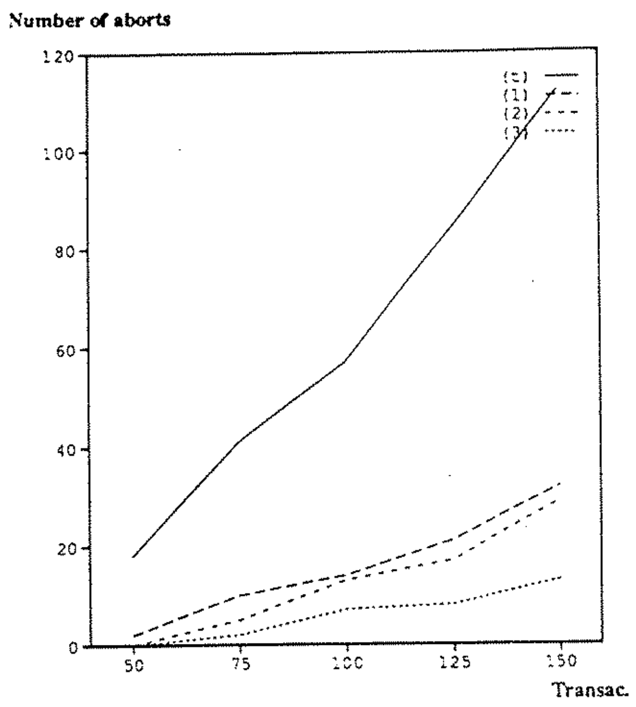

Figure 6a: SITES = 3

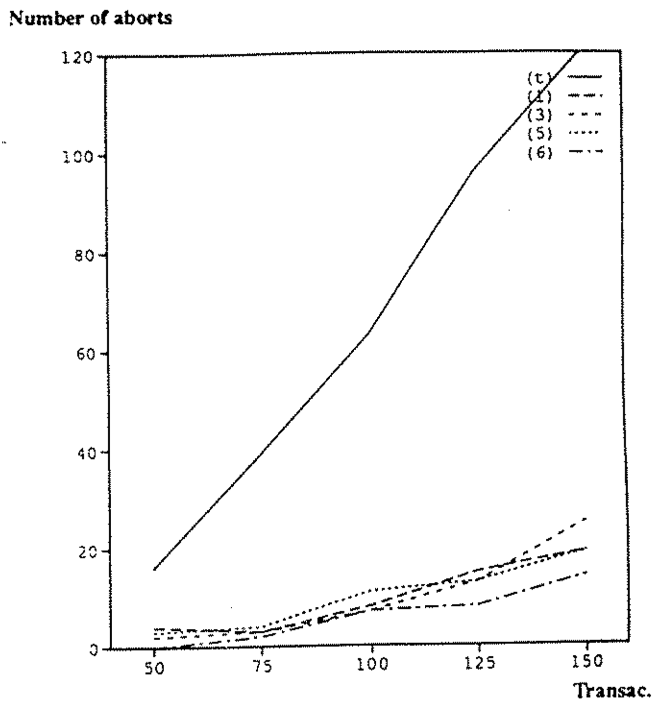

Figure 6b: SITES = 6

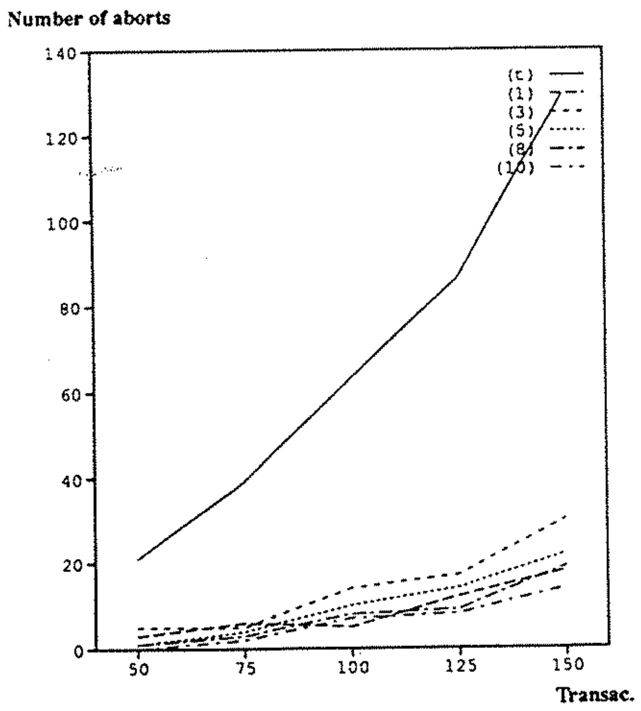

Figure 6c: SITES = 10

Throughput = F(mean interarrival time of transactions)
Transactions: 100
Communication cost: 1ms
Read only transactions: 40%
Transaction size: 12 data objects

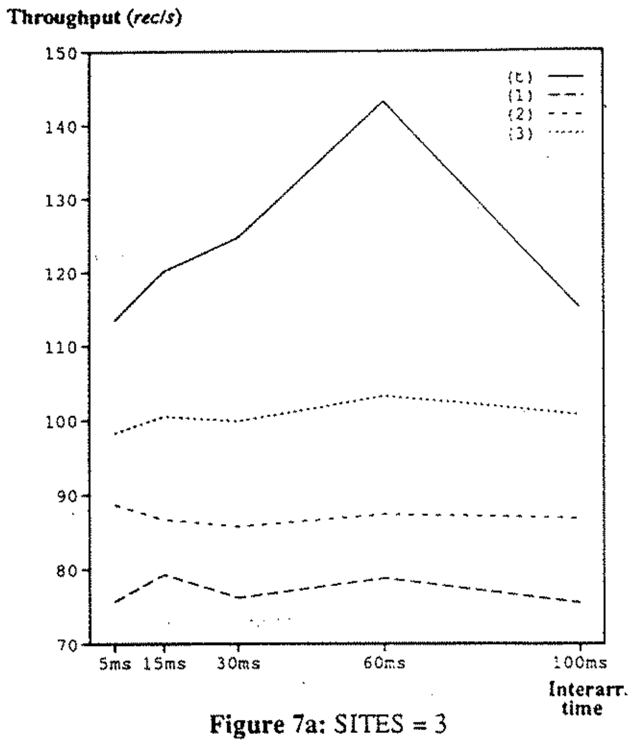

Figure 7a: SITES = 3

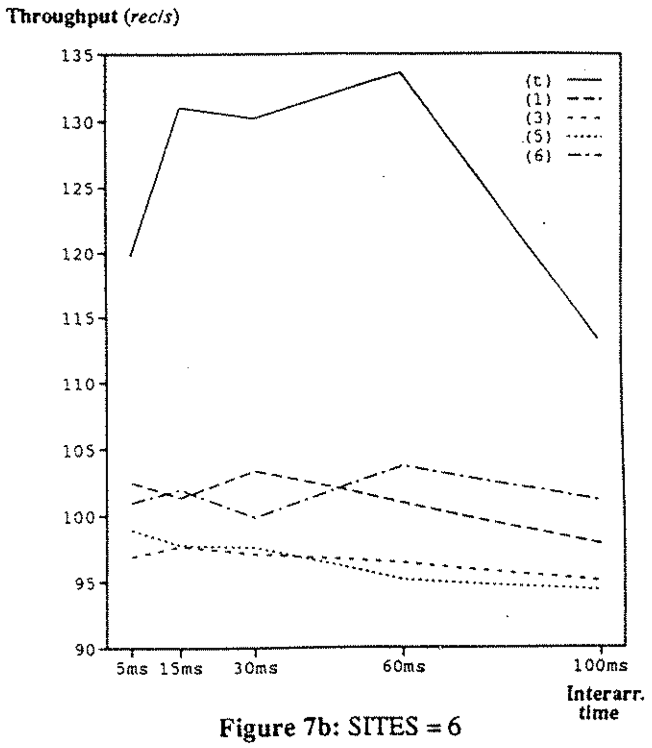

Figure 7b: SITES = 6

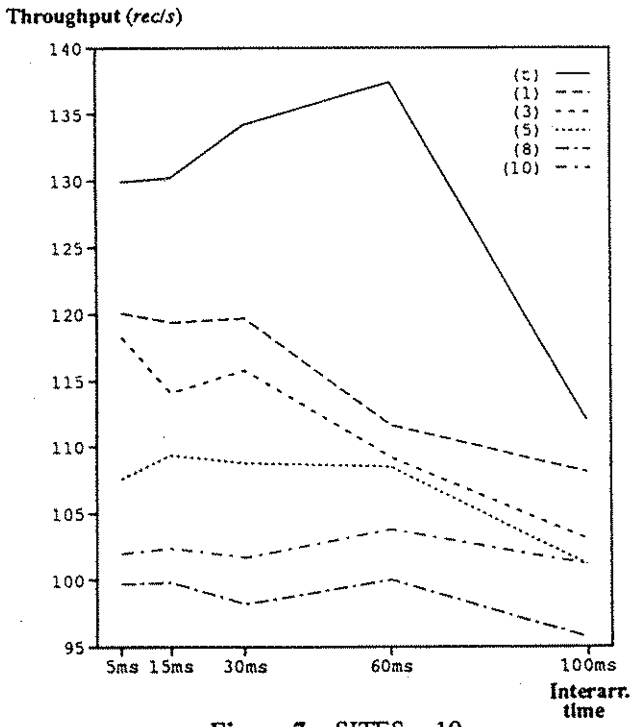

Figure 7c: SITES = 10

Response time = F(mean interarrival time of transactions)
Transactions: 100
Communication cost: 1ms
Read only transactions: 40%
Transaction size: 12 data objects

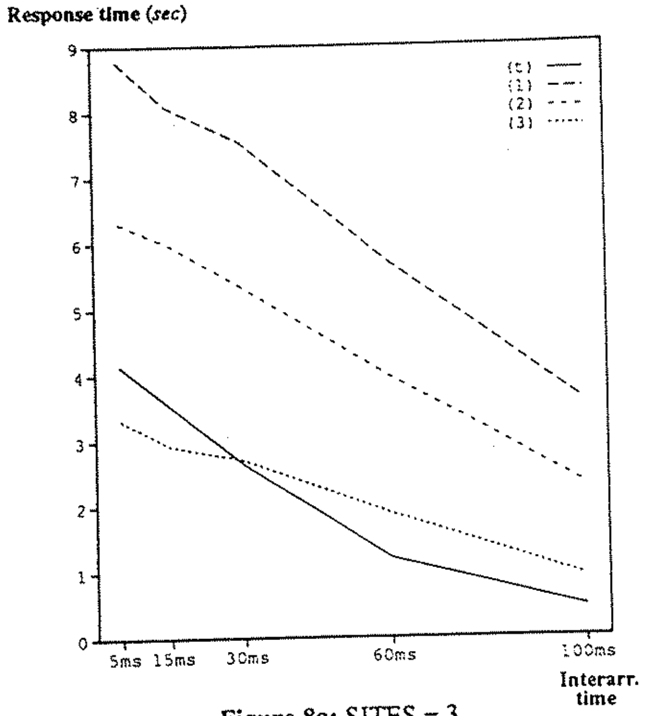

Figure 8a: SITES = 3

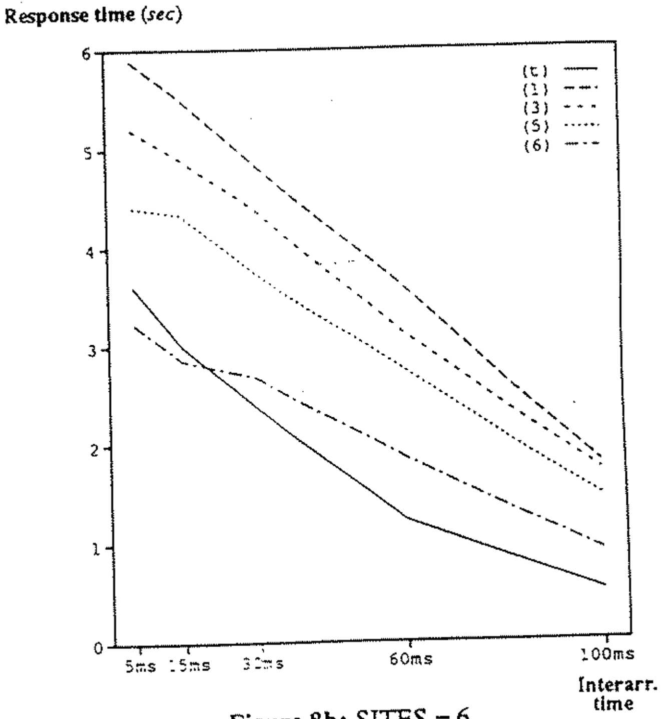

Figure 8b: SITES = 6

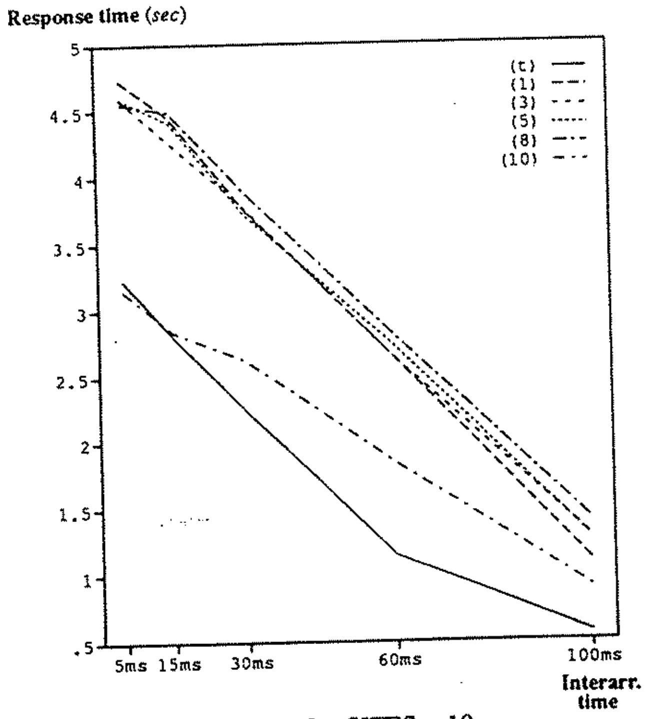

Figure 8c: SITES = 10

Throughput = F(communication cost)
Transactions: 100
Interarrival time: 30ms
Read only transactions: 40%
Transaction size: 12 data objects

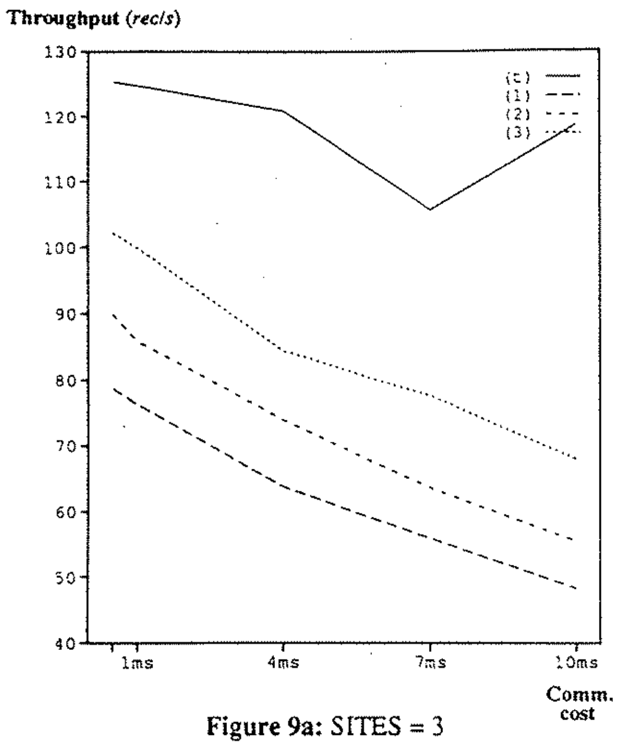

Figure 9a: SITES = 3

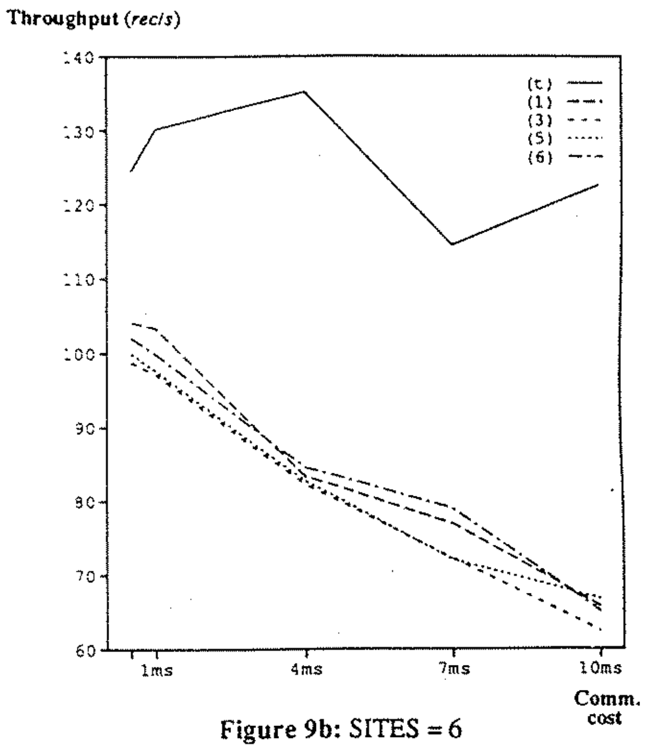

Figure 9b: SITES = 6

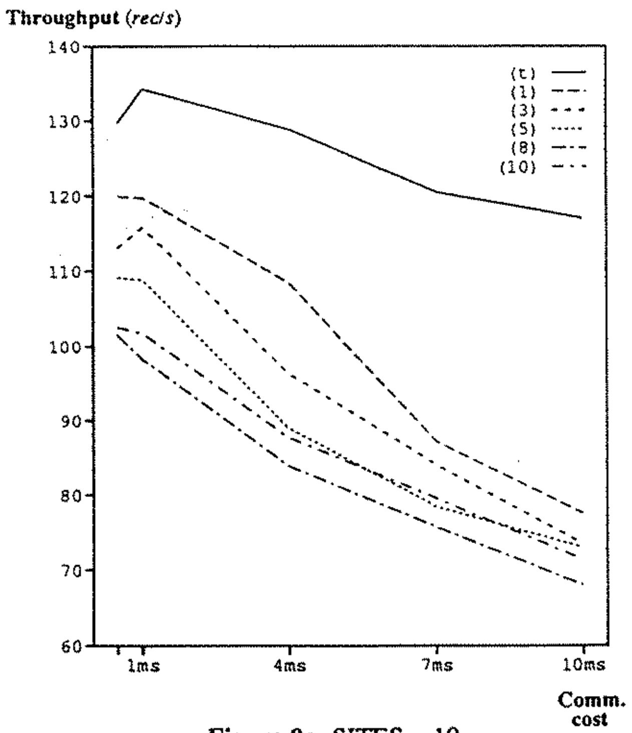

Figure 9c: SITES = 10

Response time = F(communication cost)
Transactions: 100
Interarrival time: 30ms
Read only transactions: 40%
Transaction size: 12 data objects

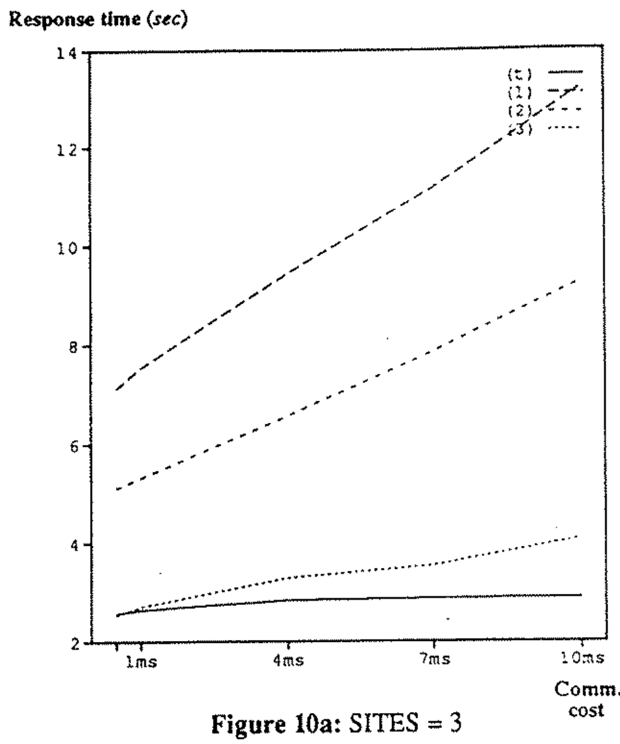

Figure 10a: SITES = 3

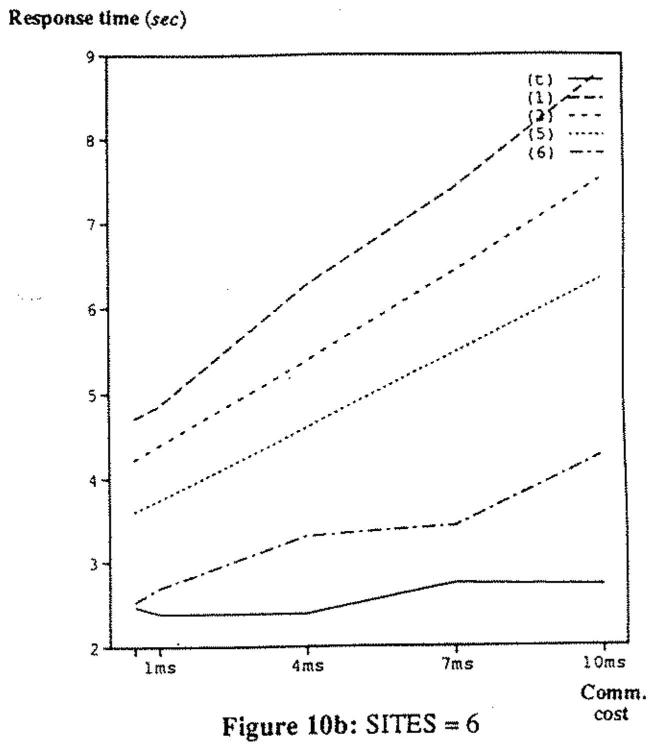

Figure 10b: SITES = 6

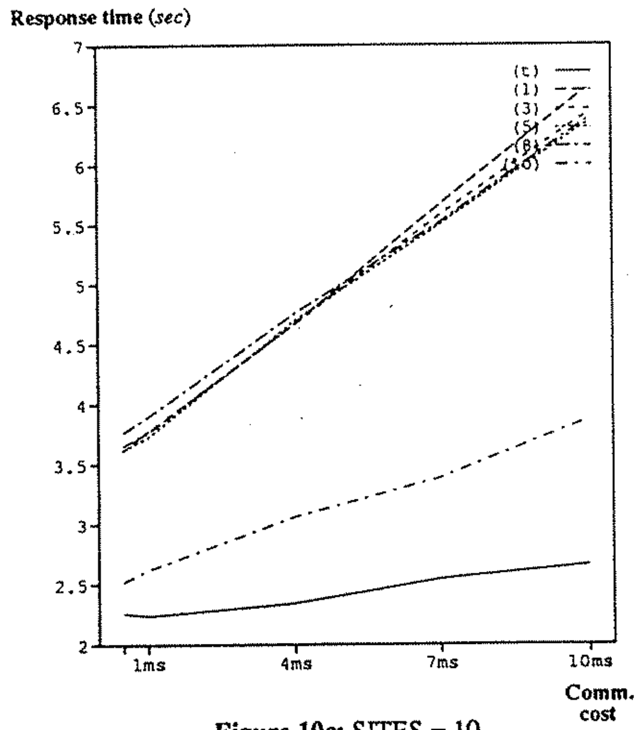

Figure 10c: SITES = 10

Throughput = F(percentage of read only transactions submitted to the system)
Transactions: 100
Interarrival time: 30ms
Communication cost: 1ms
Transaction size: 12 data objects

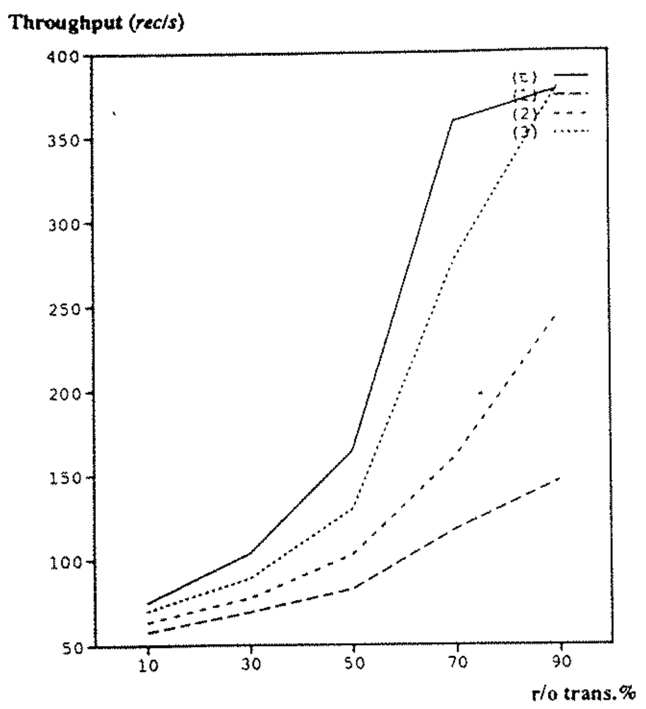

Figure 11a: SITES = 3

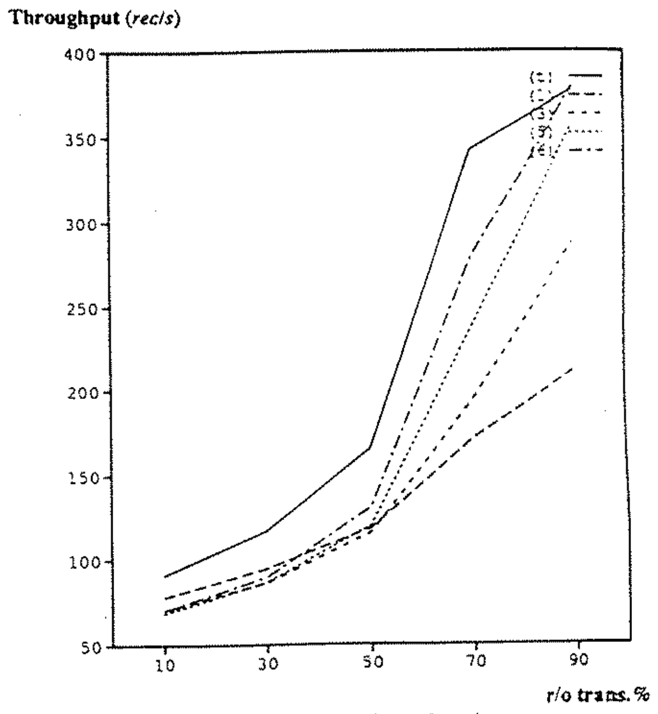

Figure 11b: SITES = 6

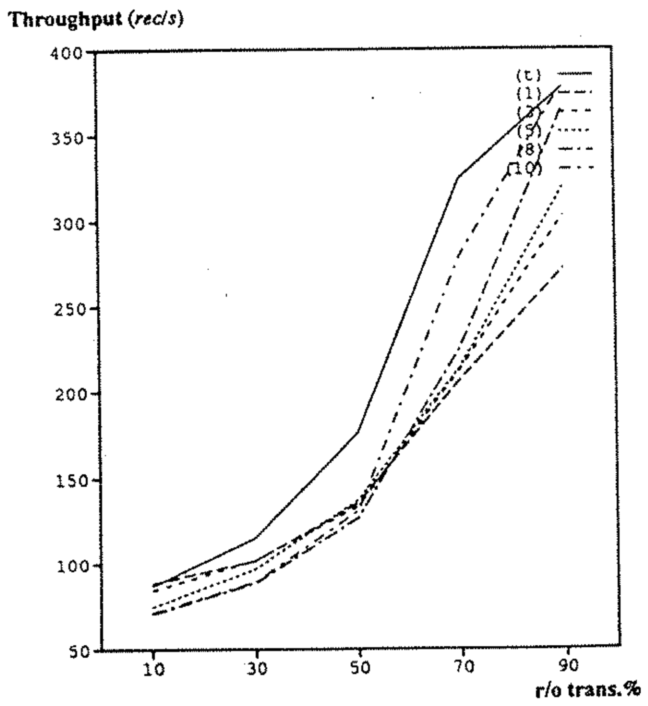

Figure 11c: SITES = 10

Response time = F(percentage of read only transactions submitted to the system)
Transactions: 100
Interarrival time: 30ms
Communication cost: 1ms
Transaction size: 12 data objects

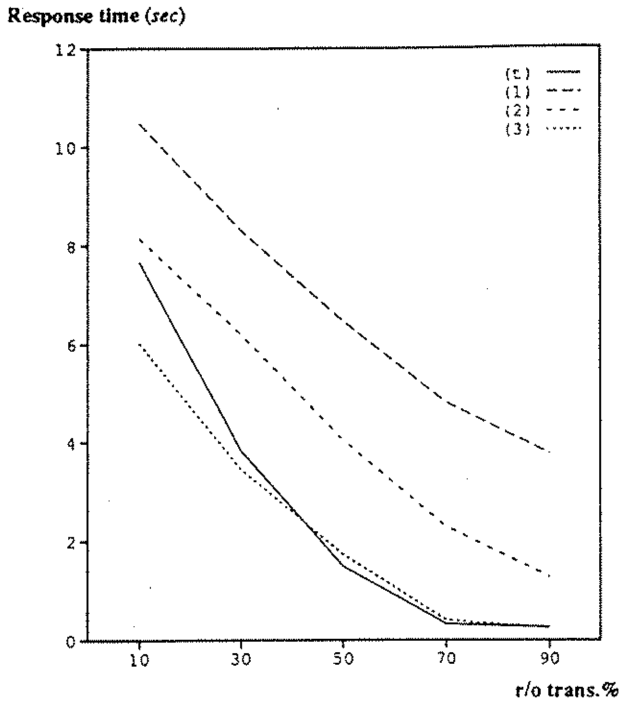

Figure 12a: SITES = 3

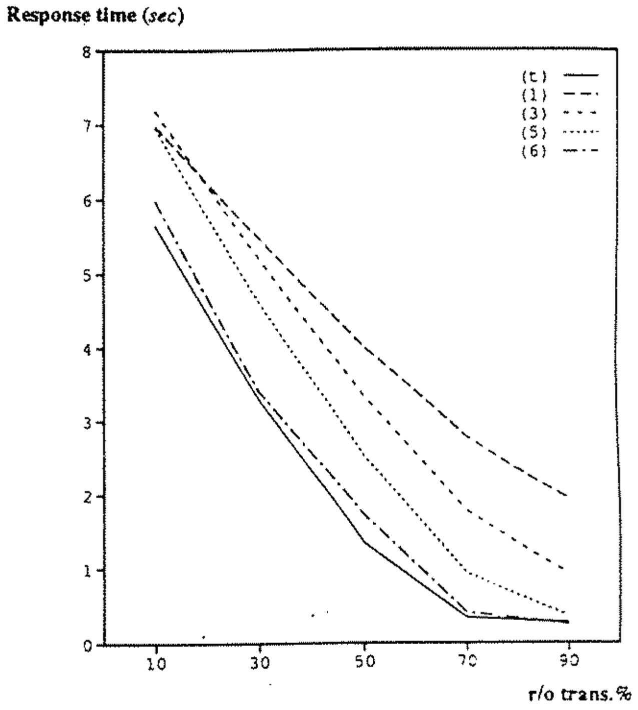

Figure 12b: SITES = 6

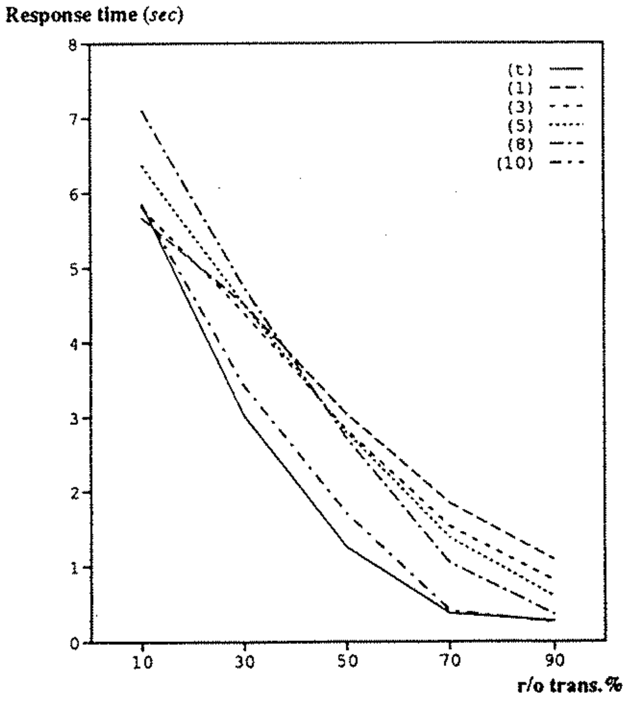

Figure 12c: SITES = 10

Total number of transaction aborts = F(percentage of read only transactions submitted to the system)
Transactions: 100
Interarrival time: 30ms
Communication cost: 1ms
Transaction size: 12 data objects

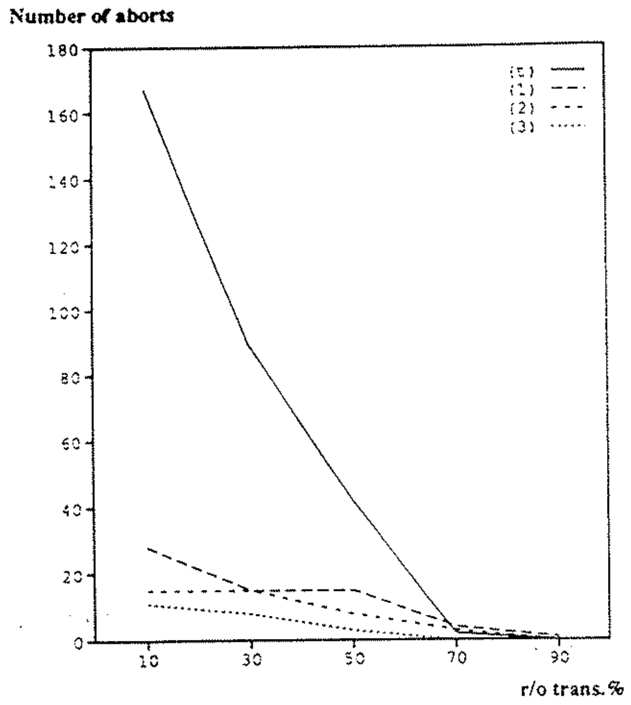

Figure 13a: SITES = 3

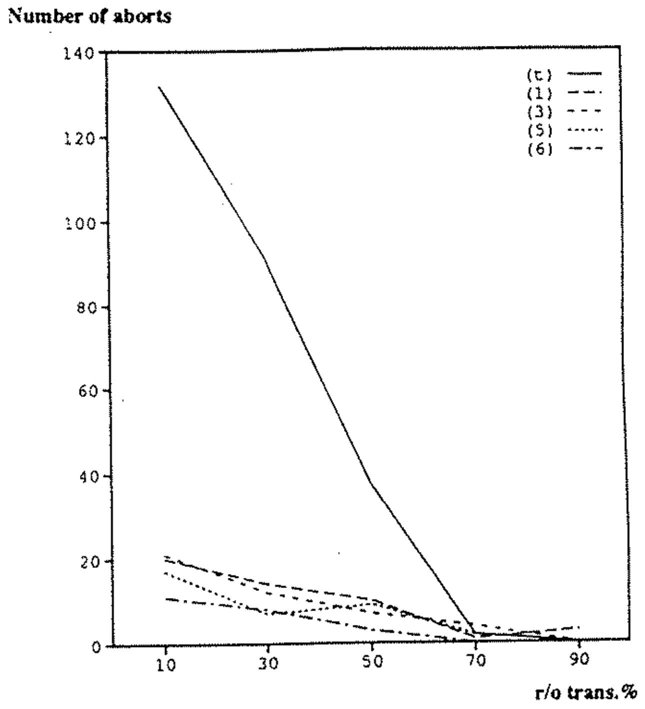

Figure 13b: SITES = 6

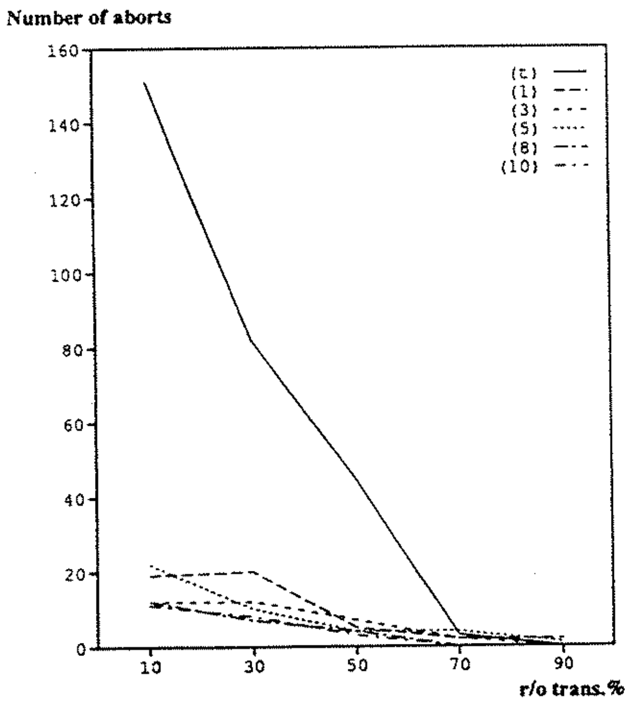

Figure 13c: SITES = 10

Throughput = F(abort cost per transaction)
Interarrival time: 30ms
Communication cost: 1ms
Transactions: 100 (r/o transactions = 40%)
Transaction size: 12 data objects

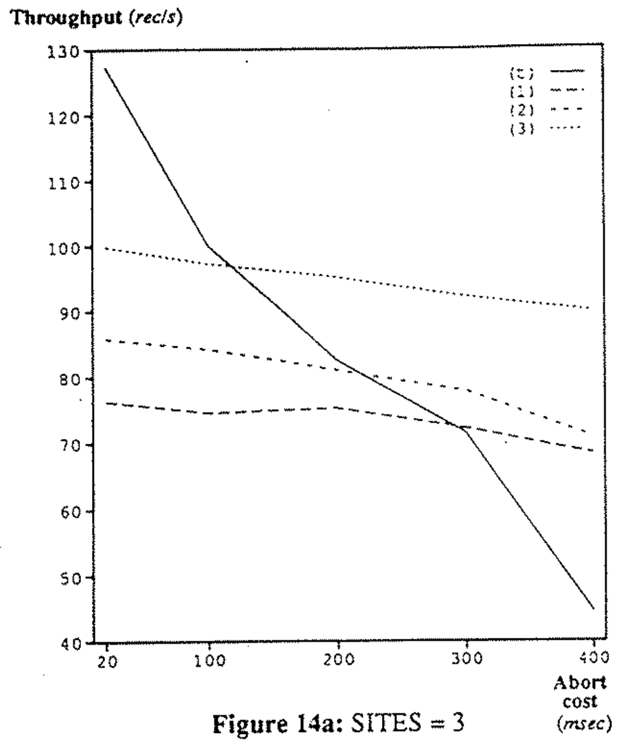

Figure 14a: SITES = 3

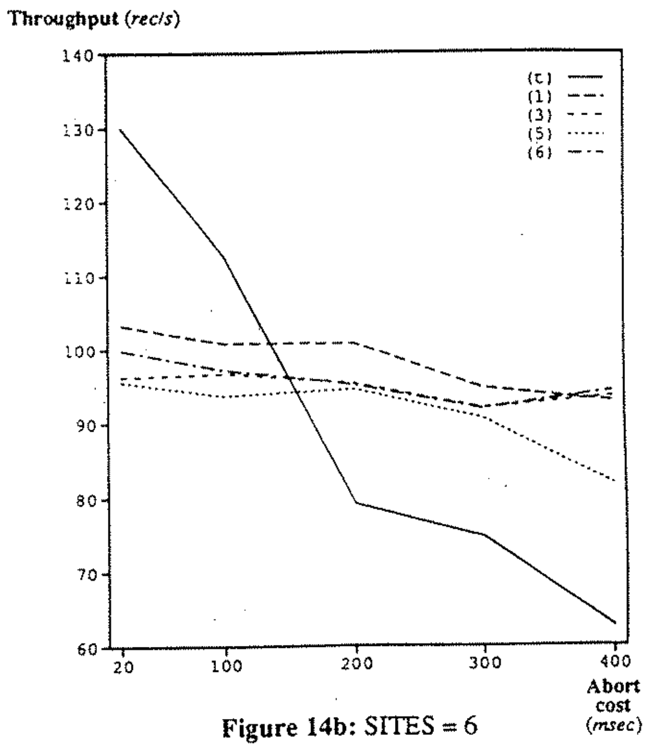

Figure 14b: SITES = 6

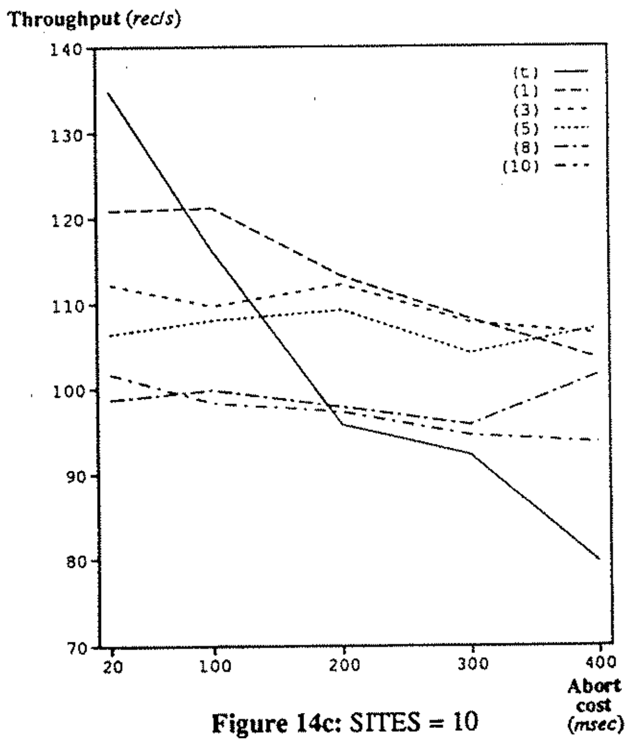

Figure 14c: SITES = 10

Response time = F(abort cost per transaction)
Interarrival time: 30ms
Communication cost: 1ms
Transactions: 100 (r/o transactions = 40%)
Transaction size: 12 data objects

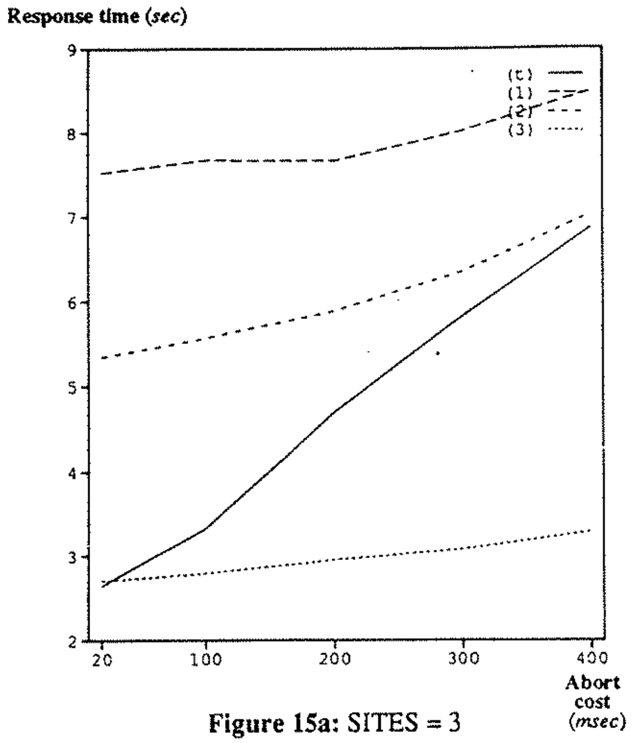

Figure 15a: SITES = 3

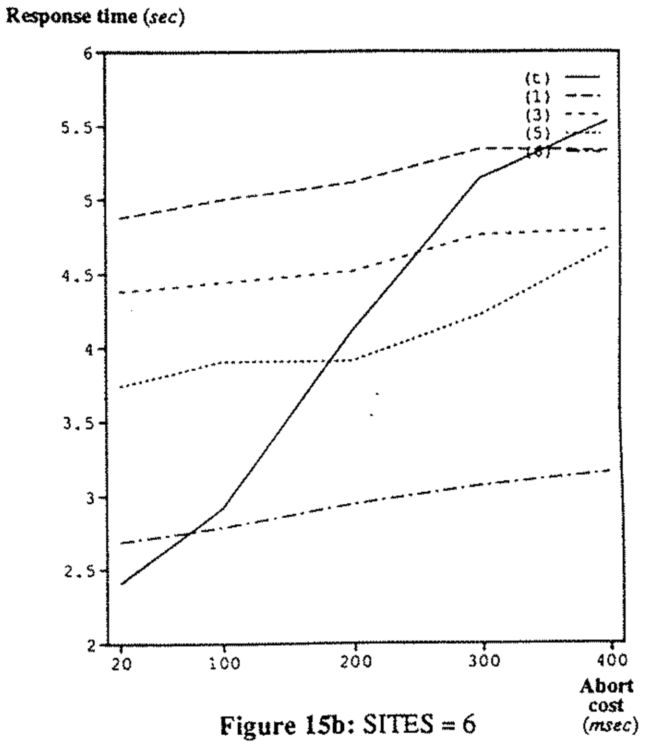

Figure 15b: SITES = 6

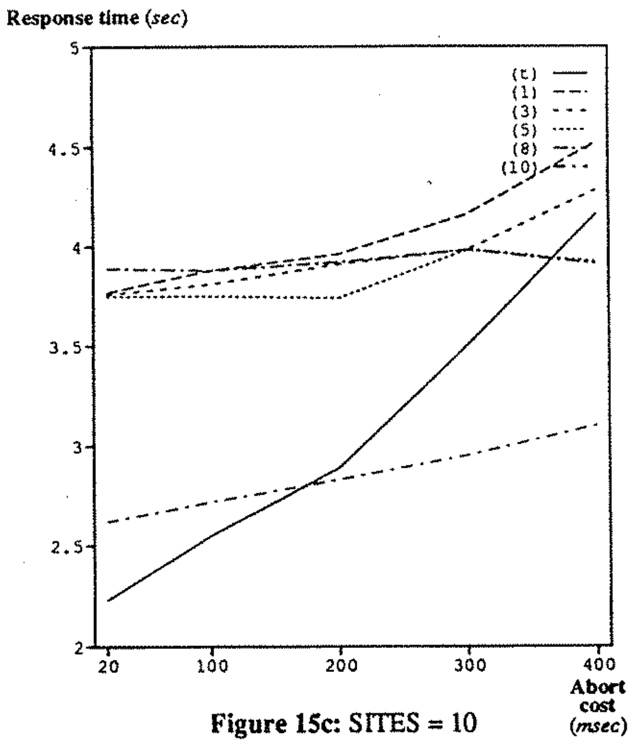

Figure 15c: SITES = 10

REFERENCES

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

[2] D.Barbara and H.Garcia-Molina, "Evaluating Vote Assignments with a Probabilistic Metric", Digest of Papers FTCS-15: 15th Int. Symp. on Fault-Tolerant Computing, Ann Arbor, Michigan, pp. 72 - 77, June 1985.

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

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

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

[6] 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.

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

[8] 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.

[9] M.J.Carey, R.Jauhari, and M.Livny, "Priority in DBMS Resource Scheduling", Proceedings of the 15th VLDB Conference, Amsterdam, 1980.

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

[11] H.Garcia-Molina and D.Barbara, "How to Assign Votes in a Distributed System", Journal ACM, Vol. 32, No. 4, pp. 841 - 860, 1985.

[12] M.Herlihy, "A Quorum-Consensus Replication Method for Abstract Data Types", ACM Trans on Database Systems, Vol. 4, No. 1, pp. 32 - 53, 1986.

[13] T.V.Lakshman and D.Ghosal, "A New Symmetric O(√N) Multiple Copy Update Algorithm and its Performance Evaluation", Tech.Rep. CS-TR-2502, Institute for Advanced Computer Studies, Un. of Maryland, July 1990.

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

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

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

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

[18] 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.