Real-Time Processing in Client-Server Databases *

Vinay Kanitkar† and Alex Delis
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201

November 2000

Abstract

In this paper, we propose and experimentally evaluate the use of the client-server database paradigm in real-time processing. To date, the study of transaction processing with time constraints has been mostly restricted to centralized or "single-node" systems. Recently, client-server databases have exploited locality of data accesses in real-world applications to successfully provide reduced transaction response times. Our objective is to investigate the feasibility of real-time processing in a data-shipping client-server database architecture. We compare the efficiency of the proposed architecture with that of a centralized real-time system. We discuss the transaction scheduling issues in the two architectures and propose a new policy for scheduling transactions in the client-server environment. This policy assigns higher priorities to transactions that have a greater potential for successful completion by harnessing locally available data. Through a detailed performance scalability study, we investigate the effects of client data-access locality and various updating workloads on the transaction completion rates. Our experimental results show that real-time client-server databases can provide significant performance gains over their centralized counterparts. These gains become evident when large numbers of clients (more than forty) are attached per server, even in the presence of very high data contention.

1 Introduction

The proliferation of network-centric computing has created opportunities for corporations and institutions to support their business cycle with large-scale financial information systems. Inter-agency transactions, known collectively as electronic commerce (E-commerce) [11, 32], will lead to new models of interaction among organizations and individuals. In this environment, the movement/transfer of funds and equity is performed electronically and needs to be completed within predefined deadlines. If implemented effectively, such execution of commercial operations offers a cost-effective means of managing global financial services. Modern commercial/financial applications often require that the duration of their transactions obeys time-constraints in varying degrees and they are based on the efficient interaction among distinct computing systems. Real-Time Systems (RTS) and Client-Server Databases (CS-DBSs) are two key areas that can assist in achieving the above goal. Although, there has been a great deal of independent research in these two areas, their aggregation has not been investigated at all.

*This work was supported in part by the National Science Foundation under Grant NSF IIS-9733642 and the Center for Advanced Technology in Telecommunications, Brooklyn, NY.

†Presently at Akamai Technologies Inc., Cambridge, MA 02139.

In a real-time system, CPU-bound tasks submitted to the system for execution have deadlines imposed by the application requirements. By utilizing elegant scheduling techniques and exploiting a-priori knowledge of the nature of the tasks [30, 1, 23, 39, 12], RTSs have been designed so that the time constraints on individual jobs are met with as small a percentage of missed deadlines as possible. On the other hand, client-server databases have been designed to take advantage of rapid improvements in computing power and improved network data transfer capabilities in order to provide high throughput rates. This performance enhancement has been derived by effectively harnessing the resources available in a network of clients [44, 45, 7]. While algorithms for scheduling real-time tasks [1, 16, 30, 19], real-time multiprocessor systems [31, 6, 28], and the assignment of tasks in distributed environments [23] have been addressed, there has been no coverage of real-time issues in the popular client-server (CS) database paradigm [8, 10, 24, 43]. In this paper, we make the case for real-time transaction processing in a CS-DBS environment and provide indicators of the performance and scalability of such systems. Our objective is not just to offer yet more scheduling algorithms but rather to establish through experimental means the viability and the usefulness of the proposed framework.

In real-time database systems (RTDBS), transactions –that involve non-negligible I/O operations– are scheduled with constraints on their required completion time. Time constraints are usually specified in the form of deadlines. A deadline is the latest possible time by when a transaction must complete in order to be useful. Therefore, an important measure of the effectiveness of a RTDBS is the percentage of transactions that are completed within their specified deadlines. In this paper, we investigate the performance of a CS-DBS in the presence of real-time tasks. The aggregation of these two areas is termed Client-Server Real-Time Databases (CS-RTDBSs). In the proposed framework, we use the resources available at the client sites and exploit inter-transaction data caching in order to support real-time processing. A centralized server-based real-time database system (CE-RTDBS) is used as the baseline case. This baseline is used to determine the operational parameters and workloads under which a CS-RTDBS configuration can offer promising performance results. We chose not to use a distributed real-time database system as the baseline for two reasons: (i) most current implementations of real-time databases are centralized, and (ii) in the absence of transaction-shipping between multiple database nodes, the flexibility of a distributed database system is limited.

Although client sites in this paper often seem to be directly accessible by users that should not be necessarily always the case. A formed cluster –between the client sites and their supporting server- can be used as the "nucleus" system configuration running applications on a dedicated and/or a virtual private network (VPN). User submit their transactions/requests from the "periphery" of the nucleus. Requests are handled by client sites that demonstrate physical proximity to users and/or load balancing and sharing considerations. The proposed CS-RTDBS configuration can be used to facilitate the effective development of a wide range of application systems. This set of application software includes:

1. Highly-Available Database Services: such database systems make up the core of many telecommunication operations and their goal is to not only manage voluminous data in real-time conditions with virtually zero-downtime but also to provide customers with advanced billing options and services. Data fragmentation and

a shared-nothing approach has been proposed as a way to develop such services [42]. Real-time transaction scheduling in such environments is the focus of this paper.

2. Multimedia-server architectures: the storage of a very large number of multimedia sources can be only accommodated by multiple cooperating servers. The latter can respond to a wide variety and changing workloads while complying with pre-specified quality of service requirements. In this context, new strategies for data placement for video/audio applications operating in variable bit rate and 3-D interactive virtual worlds are matters of recent proposals [35].
3. Ultra-fast Internet content delivery: there exist multiple concurrent efforts that attempt to bring web content closer to the requesting user. This can be either by using multiple proxies or content facilities around the globe and trying to always furnish "updated" content. These facilities could be implemented as clusters of sites with characteristics similar to those advocated in this paper.
4. Efficient access for massive e-commerce user communities: in seeking ways to ensure that at all times there are available resources to service user requests, prominent retailers and corporations plan/implement backbone multi-server systems capable of isolating classes of requests [15, 2, 3, 5]. In doing so, organizations can overcome overload and response delays. The development of such multi-server farms can follow our system model.
5. WAP-related infrastructure: the Wireless Access Protocol was developed as a mechanism to enable access of information services (resident on the WWW and the Internet) from a wide range of PDAs and phones with limited computing resources [38, 13]. The core of this infrastructure is the WAP proxy whose role is to translate HTML language to WAP Markup Language WML is fashioned close to XML. The immense worldwide existing user community of cellular phones and PDAs can pose serious performance bottlenecks to WAP proxies; we believe the latter can be removed by deploying proxies following our proposal in this paper.

Through the development of operational prototypes, we investigate the behavior of the client-server and centralized RTDBS configurations in a number of diverse settings. We evaluate the prototypes of the two configurations with contrasting workloads that feature mixed types of transactions with gradually varying degrees of update selectivities. We show that overall the CS-RTDBS is much more scalable than its centralized counterpart as the transactional load on the system increases.

We have organized this paper as follows: in the next section, we describe models of the CE and CS-RTDBS configurations as well as the policies used to schedule transactions with time constraints. Section 3 provides detailed information about the development of our system prototypes. Experimental parameters and the results of our performance study are presented in Section 4. Section 5 briefly discusses related work. Conclusions can be found in Section 6.

2 RTDBS Configurations

In this section, we describe the processing models of the two architectures and explain the transaction scheduling algorithms used.

2.1 Models and Assumptions

This subsection describes the transaction processing models of the centralized and CS database architectures. Here, we assume that the database is a collection of uniquely identifiable objects. In the centralized architecture (Figure 1), CE-RTBDS, the database server performs all the transaction processing. The clients in such a system are assumed to be simple terminals and serve as user-interface devices only. Clusters of clients are managed by a terminal server which handles all communication between the terminals (clients) and the centralized server. Transactions are initiated at the clients and are immediately transported to the server for execution via an IPC abstraction. There, the transactions are scheduled by a single centralized scheduler according to some priority assignment algorithm. Since concurrent execution of transactions is possible, a locking protocol is used to serialize access to database objects. The results of executing the transactions are communicated to users through their terminals. This is an early form of the query-shipping approach [34].

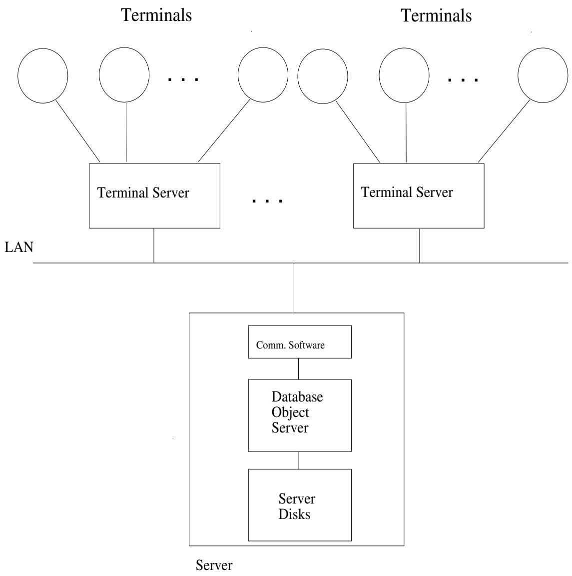

Figure 1: The Centralized Database Model

In a data-shipping CS architecture [29, 33], client sites are expected to have significant processing and storage capabilities (Figure 2). Transactions are initiated, scheduled, and executed at client sites. The locally available buffer space and CPU of a client is used to carry out the necessary high-level database processing. This includes application-specific presentation management, query processing, and transaction management. If a database object referenced by a transaction is not cached locally then it must be first fetched from the server before it can

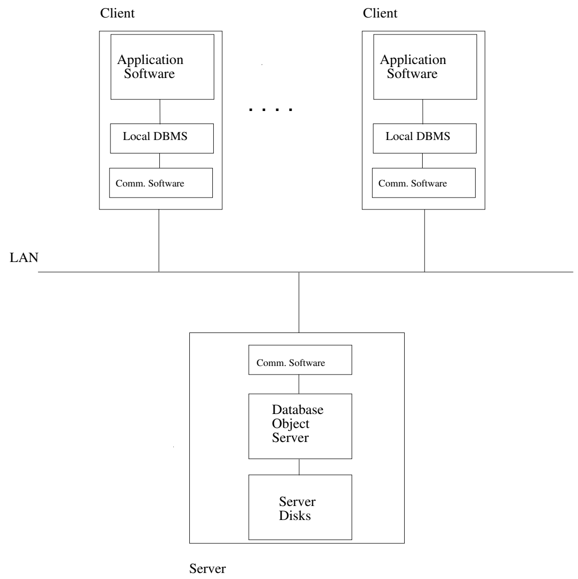

Figure 2: The Client-Server Database Model

be used. The server ships this object to the client and the transaction that requested the object is executed locally. Thus, all the transaction processing is actually performed at the clients. The server performs only low-level database functionalities (I/Os, buffering and management of concurrency) on the behalf of requesting clients. The set of objects cached at a client's disk and memory buffers is treated as a local data-space. Objects/locks that have been fetched from the server are returned to the server only if the latter explicitly calls them back. This type of inter-transaction caching allows future requests on the cached objects/locks to be satisfied without any interaction with the server. However, inter-transaction caching of data also complicates database recovery in case of client failures. The durability of committed client transactions after a client crash needs to be ensured. Although, we consider database recovery to be beyond the scope of this paper, we envision the possibility of extending an algorithm like ARIES/CSA [27] to this environment in the near future.

Global concurrency control is based on pessimistic locking and is performed by the server using a lock table. Clients are permitted to hold two types of locks, shared (read-only) and exclusive (read-write) [18, 29, 44]. More than one clients can hold shared locks (SL) on an object, but an exclusive lock (EL) can be granted to only one client at a time. Additionally, if a client holds an exclusive lock on an object then no other client can have any type of lock on that object. When a lock is granted to a client, a copy of the object is shipped over to the requesting client (if necessary). If another client has a conflicting lock on that object, the server then contacts that client and requests it to release the lock on the object and return the object, if necessary. This is done by the client as soon as no local transactions are accessing the object in question. Once the object has been returned, the server grants the lock to the requesting client and dispatches the object to it.

This locking mechanism has been termed as *Callback Locking* in [18, 14, 44]. It guarantees serialized access to the database, but it can also cause unnecessary blocking of transactions as they wait for the requested locks to be granted, especially in a distributed environment. To alleviate the delays caused by such blocking, we have modified the above locking mechanism. When the server requests a client to give up an exclusive lock on an object, it also specifies the type of lock the presently requesting client desires. If the competing request is for a shared lock then the client that holds the exclusive lock ships a copy of the updated object to the server (as soon as the transaction that is currently using the object is complete) but only *downgrades* its own lock to a shared lock. Now the server grants a shared lock to the requesting client and ships the object to it. Transactions at both clients can now access the object in a shared fashion. We believe that this novel technique, which we call *Enhanced Callback Locking*, will permit greater data sharing among multiple sites, especially in low update situations. The server's global lock table is also used to maintain a wait-for graph that stores granted and outstanding lock requests. This wait-for graph is used to detect global deadlocks caused by clients' object requests.

- Transaction T requests object A from the client's local lock manager.
- IF(object A is available in the local cache with the appropriate lock -- shared or exclusive) THEN
  - IF(no other concurrent local transaction has locked the object in a conflicting mode) THEN
    * Transaction T is granted the lock and allowed to access the object.

    ELSE
    * T blocks until the transaction(s) holding the lock(s) commit. Now, T is granted its requested lock on A, and allowed to continue with its processing.

    ENDIF

  ELSE
  - Request for the appropriate lock is dispatched to the server.
  - T is blocked until the server grants the requested lock. Usually, when the server grants a lock it also sends the object over to the client. The only exception is for exclusive lock requests when the object is already present at the client with a shared lock. In this case, the server only needs to grant the lock.
  - Once the lock is granted by the server, transaction T is granted the requested lock and allowed to proceed.

  ENDIF
- When transaction T commits, it releases all the locks it has acquired. The objects that have been fetched from the server are held in the client's cache along with the granted locks.

Figure 3: Processing of Object Requests by Client Transactions

In the CS-DBS, each client also has a local lock manager [26]. Since each client can execute multiple transactions concurrently, it is necessary for all transactions to acquire appropriate locks on the objects they access. Transactions are allowed to acquire shared or exclusive locks on locally cached objects but, the lock that can be granted by the local lock manager depends on the lock that the client has acquired from the server. Therefore, a client transaction cannot acquire an exclusive lock on a cached object if the client itself has only acquired a shared lock from the server. This hierarchical, strict locking policy ensures that conflicting locks cannot be simultaneously granted to client transactions anywhere in the cluster. The precise algorithm used by the clients for executing local transactions is given in Figure 3.

Clients' lock managers also maintain up-to-date wait-for graphs that store granted and outstanding requests from local transactions. The local wait-for graphs are used to detect and resolve deadlock cycles among local transactions. Deadlock detection at the clients is performed independent of the global deadlock detection performed by the server. This is due to the fact that each transaction is executed at precisely one client site (in a non-distributed manner), and therefore, all the necessary objects/locks have to be cached at that client.

In this paper, we suggest the use of such a client-server database architecture in real-time transaction processing environments. As the CS model maintains improved response times in the presence of spatial and temporal locality of client data accesses, we believe it can lead to more effective real-time processing. Once data objects are available at the client site, the local scheduler can independently determine the order of execution among the pending transactions which no longer need to access server data. The next section discusses a number of algorithms used to schedule tasks either at the client or the server site depending on the configuration used (CS-RTDBS and CE-RTDBS).

2.2 Transaction Scheduling Algorithms

In the context of real-time database systems, transactions are assumed to have deadlines associated with them. A transaction successfully completes only if it has finished executing within its deadline. In a RTDBS, the priority assignment used to schedule transactions often plays an important part in deciding the efficiency of the system. This efficiency is measured in terms of the percentage of transactions completed within their deadlines. For each transaction, there are three basic pieces of information available: the transaction's arrival time, deadline and expected processing time. This information can be used in several ways to assign priorities to transactions [1, 21]. Two of the most commonly used policies are: Earliest Deadline First (ED), and Least Slack First (LS). In addition to these two scheduling algorithms, we also experiment with the First-Come First-Serve policy FCFS schedules transactions to be processed in the order in which they arrive. Since the deadline information about the transactions is not used, FCFS will discriminate against a newly arrived transaction with an earlier deadline in favor of an older transaction which may have a later deadline. Therefore, FCFS is not a real-time scheduling policy and we use it only as a baseline technique for comparison with other scheduling policies.

Earliest Deadline First scheduling utilizes the deadline information of the transactions. In this regard, the transaction with the earliest deadline is assigned the highest priority. This scheduling policy does not look at the expected processing time required for transactions, and hence, has the weakness that it can schedule transactions that have missed their deadline or are certain to.

Least Slack First defines a slack time for each transaction that is computed using the formula: $S = d - (t + E)$, where t is the current time, and E and d are the estimated processing time and deadline for the transaction respectively. The slack time, S, is an estimate of how long a transaction can be postponed without missing its deadline. The transaction that has the least slack time will be scheduled first by this algorithm. A negative slack time indicates that a transaction has missed its deadline. The efficiency and usefulness of Least Slack First scheduling depends heavily on accurate knowledge of transaction processing times. In our discussion of this policy, we make an important assumption that the CPU processing time of all transactions is known precisely. Note that we make no assumptions about the time a transaction could have to wait for its requisite data to become available. Therefore, it is quite possible that a transaction could fail to meet its deadline because it could not gain access to appropriate data objects.

In addition to these scheduling strategies, only transactions that have feasible deadlines (*FD*) are selected for execution. This ensures that transactions that are expected to miss their deadlines are not processed at all. Using this measure to determine whether a transaction should be processed requires reasonably accurate estimates of its execution time. Transactions that have already missed their deadlines (tardy transactions) can easily be abandoned or aborted. For simple examples of how these scheduling policies prioritize the submitted transactions, consider the job arrival sequence shown in Table 1. Although, in a real database system, transactions are not executed one at a time, these examples allow us to demonstrate the strengths and weaknesses of the various scheduling policies.

Scheduling the jobs in the order that they arrive (pure FCFS) results in the schedule shown in Figure 4. Only transactions A, B and D meet their respective deadlines while the other three jobs are unsuccessful. In addition to FCFS scheduling, processing only those transactions that have not yet missed their deadlines (Not Tardy or *NT*) results in the schedule depicted in Figure 5. In this schedule, at time 6, Job C has missed its deadline and, therefore, Job D is picked for processing. When Job D finishes at time 14, Transaction F is processed because E has already become tardy. Since FCFS with NT is clearly better than Pure FCFS, we do not examine pure FCFS in the rest of the paper. Instead, we use the term FCFS to mean FCFS with NT.

Job	*Arrival Time*	*Processing Time Required*	*Deadline*
A	0	2	3
B	1	4	10
C	2	2	5
D	5	8	16
E	7	2	13
F	11	3	19

Table 1: An Example Set of Job Arrivals

Using the ED scheduling algorithm, results in the schedule shown in Figure 6. Here too, the NT criteria is used so that no CPU time is wasted in processing transactions that have already missed their deadlines. At time 0, A is the only job in the system so it is processed immediately. At time 2, there are two jobs available, B and C. The

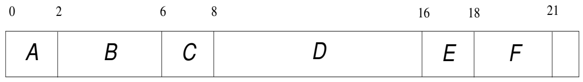

Figure 4: Pure FCFS Scheduling

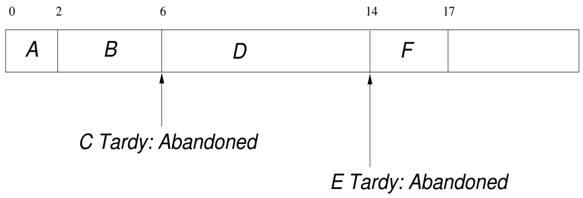

Figure 5: FCFS Scheduling with NT

deadline for B (10) is later than that for C (5) so Job C is selected for processing at this time. Similarly, at time 8, Job E has an earlier deadline that Job D, so E is processed first.

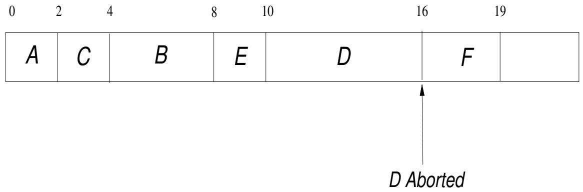

Figure 6: Earliest Deadline First Scheduling with NT

If the Least Slack First algorithm is used, the schedule that is generated is shown in Figure 7. At time 2, either of B and C have to be selected for processing. The slack for Job B is 4 [10 - (2 + 4)] and the slack for Job C is 1 [5 - (2 + 2)], hence Job C is processed first. In the same manner, at Time 8, Task D has a slack 0 [16 - (8 + 8)] and Task E has an available slack of 3 [13 - (8 + 2)], and hence, D is processed before E. At Time 16, E has a negative slack (-5) that indicates that E has missed its deadline. Therefore, Job F is processed at this time.

As the above example demonstrates, pure FCFS is virtually assured to do worse than the other scheduling policies. However, it is not easy to theoretically compare the schedules generated by FCFS (with NT), ED (with NT), and LS (with FD and NT). This is because the quality of the generated schedules depends largely on the actual job arrival times and deadlines. It has been shown experimentally that ED and LS perform better than FCFS under most situations except when the transactional load is very high [1].

The algorithms described above can be used to prioritize tasks either at the server in a centralized or at the clients in a client-server setting. The availability of cached data in a CS-RTDBS can play an important role in scheduling the execution of client-based transactions. In the absence of its necessary data, a higher priority transaction may have to be scheduled after a lower priority one if the latter has all its required data/locks cached at the client. Also, in our previous work on CS-RTDBSs [22], an important observation we made with regard to failed transactions was that the time spent by a such transactions waiting for the appropriate locks on their requested objects played a major part in their failure. Therefore, we propose a scheduling policy that makes its scheduling decisions on

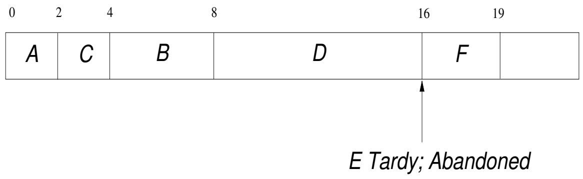

Figure 7: Least-Slack First Scheduling with FD and NT

the basis of the data locally available at the clients. The task that has the highest proportion of its required data available locally is assigned the highest priority. If two tasks have the same percentage of their required data available locally, then the ED policy is used as a tie-breaker. We term this scheduling strategy *Most Locally Available Data First* (LAD). For instance, consider the set of jobs shown in Table 2 which is a minor variation of that shown in Table 1. The additional information available for each job is the percentage of its so far requested objects that are already available at the site of execution. In general, the set of objects that a transaction will access during its execution is not easy to predict a priori. Although, we do not update the values for the percentage of locally available data (objects) in the given example (Table 2), in our implementation, these values are updated dynamically as object requests are fulfilled by the server and when local transactions release locks on objects that other local transactions are waiting to access. The transaction priority queue is also updated accordingly.

The order in which these jobs are scheduled is shown in Figure 8. At time 0, A is the only job available so it is started immediately. When this task finishes, at time 2, B and C are in the scheduler's queue. Of these two jobs, C is selected for execution first since it has a greater proportion of its required data available locally. At time 4, B is executed as it is the only job in the queue. Once job B completes, the choice to be made is between D and E. According to LAD scheduling, E will be picked as it has 80% of its data resident at the site as compared to 65% for D. At time 11, D is dropped from the job queue since it has missed its deadline. The decision to be made, at time 12, is between tasks F and G both of which have a deadline of 19. Since LAD uses the deadline only as a secondary measure, F is executed first because it has more of its data locally resident. G is executed after F is done, but is aborted when it misses its deadline (at time 19).

Job	*Arrival Time*	*Processing Time Required*	*Deadline*	*Percentage of So Far Requested Data Objects Locally Available*
A	0	2	3	0.8
B	1	4	10	0.8
C	2	2	5	0.9
D	5	4	16	0.65
E	7	4	13	0.8
F	11	3	19	0.9
G	12	5	19	0.7

Table 2: Example Set of Job Arrivals for LAD Scheduling

The performance of LAD is tied very closely to the degree of locality that transactions at each site in the cluster

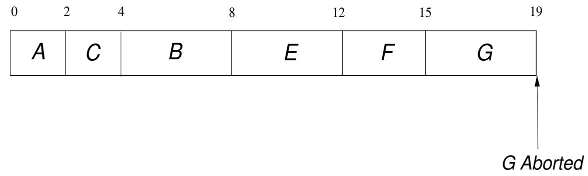

Figure 8: Most Locally Available Data First Scheduling with NT

demonstrate. A higher percentage of data requests that can be satisfied locally implies a lower percentage of missed deadlines. We believe that, in the presence of tight time constraints on tasks, the use of deadline information as a secondary heuristic will not affect the performance of LAD very significantly even when the degree of access locality is lower.

In the next section, we describe the development of the system prototypes used to evaluate the CS-RTDBS and the CE-RTDBS.

3 Development of System Prototypes

In this section, we describe the implementation of our system prototypes of the centralized (CE-RTDBS) and client-server (CS-RTDBS) architectures. The prototypes for both systems have been developed on Solaris 2.5 using the available thread and socket libraries, as well as a library for paged I/O operations. The overall size of the CS-RTDBS package is about 11.3 K lines of C code while the size of the CE-RTDBS package is approximately 7.9 K lines.

In both systems, communication between the clients and the server is done using TCP sockets. In both implementations, the server has been designed as a concurrent connection-oriented server. Once a connection has been created between a client and the server, this connection is maintained for the duration of the experiment. This is done so that the relatively high overhead of establishing a socket connection between the clients and the server is incurred only once. At the beginning of each experiment the server listens on a socket for client connection requests. As soon as a connection request arrives from a client, the server creates a new lightweight process (Solaris thread) to handle that connection. This thread interacts only with that client and handles all future requests from it.

The Paged-File (PF) layer from the MiniRel system developed at the University of Wisconsin, Madison [40], is used to create and manage an object database (at the centralized server, CS-RTDBS clients and CS-RTDBS server). The PF layer implements a file page buffer manager. Its basic data structures are the PF file page, the page buffer, the page hash table and the file table. Each PF file is implemented as an UNIX file with the PF file page forming the basic unit of organization. In our experiments, we assume that one PF page contains exactly one database object. The page buffer is used to maintain a number of PF pages in memory. Pages requested by applications are brought into the memory buffer. If a page is already in the memory buffer then the request is satisfied without having to read from disk. If a page is to be brought into a full buffer, a victim page is chosen using LRU from among the unpinned pages in memory and thrown out. Victim pages are written back to the disk only if they have been marked as *dirty*. The page hash table is provided in order to locate the buffer entry corresponding to a PF file page in the buffer. Descriptions and typical timings for commonly used PF layer operations are given in Table 3. On top of the PF layer, our database clients maintain metadata that allows them to control the overall state of their caches. This metadata includes details about the objects cached locally. When the client cache becomes full, LRU is used to select potentially unnecessary objects in the local cache and the locks on them are released. Objects that have been updated locally are shipped back to the server before they are deleted from the local cache.

PF Layer Function	*Description*	*Timings for 100,000 Operations (in μs)*		
		Average	Maximum	Minimum
PF_GetThisPage (disk)	Fetch the specified page from disk and return a pointer to the page data	3,981	482,643	146
PF_GetThisPage (memory)	Fetch the specified page from memory and return a pointer to the page data	7	5,792	5
PF_PageUnfix	Make the buffer manager aware that the page is no longer needed in the buffer	14	11,901	6
PF_PageAlloc	Allocate a page	2,067	91,711	54
PF_DirtyPage	Mark a page as dirty	15	2,335	10

Table 3: PF Layer Primitives

A transaction in each of the prototypes is constructed using calls to PF layer functions to load and update database objects from its memory buffer or UNIX disk file. The total number of objects that each transaction accesses is decided according to an Exponential distribution. A random number of objects, between 0 and this total, are requested by the transaction as soon as it is created, and the rest are requested at periodic intervals over the its processing lifetime. Objects (pages) are read into the transaction's buffer space by using the *PF_GetThisPage()* function call which retrieves the requested pages either from the disk or PF memory buffer. Objects which are to be updated are marked as dirty using the *PF_DirtyPage()* function. Such pages are automatically written back to the disk file by the PF buffer manager when the page is replaced in the buffer. The "processing" performed by each transaction is the calculation of products of random numbers until the prescribed CPU processing time has elapsed. Just before the transaction completes, it releases the locks on the database objects.

Lock managers are used to ensure that conflicting accesses to the object database are not allowed. In the CE-RTDBS, a two-phase locking protocol employed by the corresponding manager ensures serializability of transactions. In the CS-RTDBS, the global lock manager is located at the object server and arbitrates between lock requests from clients. The lock managers in the centralized and client-server systems also maintain up-to-date wait-for graphs in order to detect deadlocks [41]. In our implementations of both systems, we do not allow a transaction with a higher priority to take away locks/objects from transactions with a lower priority. This corresponds to the *Wait* lock management scheme proposed in [1]. We do this because in our experimental system, all transactions belong to the same equivalence class in terms of their importance. However, in a system where transactions have differing levels of importance, a lock management method such as *Wait-Promote* or *High-Priority* may provide a better level of performance.

In a wait-for graph, nodes represent clients and objects, and directed edges represent granted locks and outstanding requests. A granted lock is shown as an edge from the object to the client that holds the lock while an outstanding request is shown as an edge from the requesting client to the object. In our implementation, two-dimensional arrays of integers are used to maintain wait-for graphs. Positive integers denote granted locks and negative integers identify outstanding requests. This compact representation allowed us to incorporate the wait-for graphs within the lock tables resulting in considerable savings in memory space usage.

Figure 9 depicts a deadlock involving clients 1 and 3. This is a result of allowing both, the request from client 1 for object A and that from client 3 for object B. In our implementation, when an object/lock request is received, it is added to the request queue only if it does not cause a cycle in the wait-for graph. Therefore, of the two outstanding requests shown in the figure, the request that arrives later is denied. The client that issued the request is required to release the lock it holds in the present deadlock cycle and abort the transaction in question.

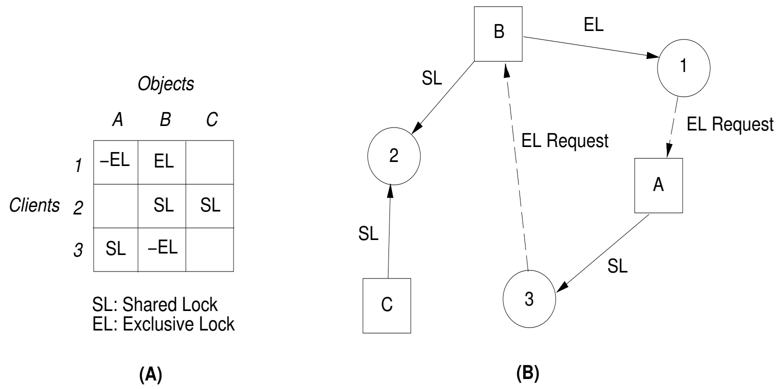

Figure 9: Using a Two-Dimensional Array to Represent a Request Wait-For Graph

In our packages, accesses to the database, lock tables, wait-for graphs and variables shared by multiple threads are synchronized by means of Solaris mutual exclusion primitives. The use of Solaris reader-writer locks was considered but their timing performance was observed to be generally poorer than the simpler mutex locking primitives in our test experiments.

The CE-RTDBS server executes one thread per client/terminal in the system. This thread maintains an open socket connection with that client for the duration of the experiment. Clients ship all transactions to the server using this socket connection. The CE-RTDBS server has been designed to be able to process a number of transactions simultaneously. This is done by executing each transaction as an independent thread. The number of transactions that can execute concurrently depends on the availability of memory buffer space and access to database objects. Figure 10 provides a pictorial representation of the server implementation. The clients (terminals) shown in the figure are simple points of service and perform no transaction processing.

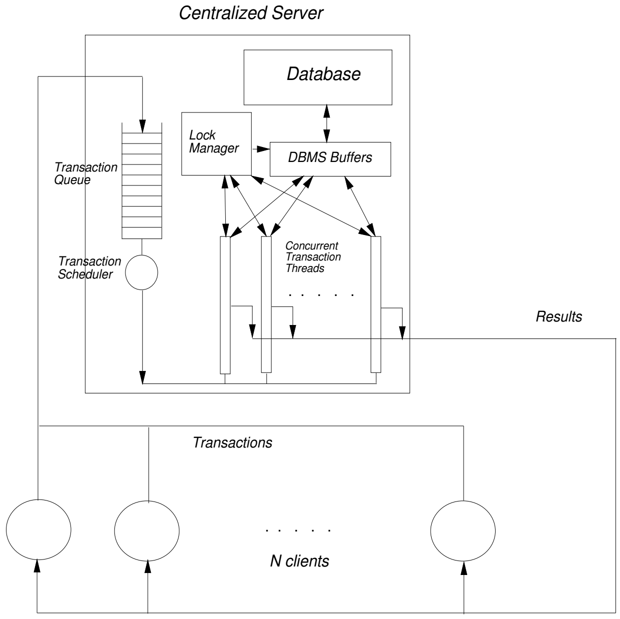

Figure 10: Implementation of the Centralized RTDBS

In the CS-RTDBS, the object-server executes one thread per client (Figure 11). Each such thread maintains two socket connections with the client for the entire duration of the experiment. One connection is used exclusively for messages while the other is used for the transfer of database objects to and from the client. The clients also make use of the short and long term memory available to them in line with the proposal in [9]. Data objects/locks that have been fetched from the server are maintained in the local cache so that future requests on cached data can be satisfied without interaction with the server.

4 Experimental Approach and Results

The objective of our experimental effort was to investigate the following points:

- What is the scalability of the centralized (CE-RTDBS) and client-server (CS-RTDBS) models as the number of clients attached to the server increase?
- How do the centralized and client-server models compare under conditions of varying update loads in the context of real-time database processing?
- What is the effect of the different scheduling policies (ED, LS, FCFS, and LAD) on the performance of the two RTDBS models?

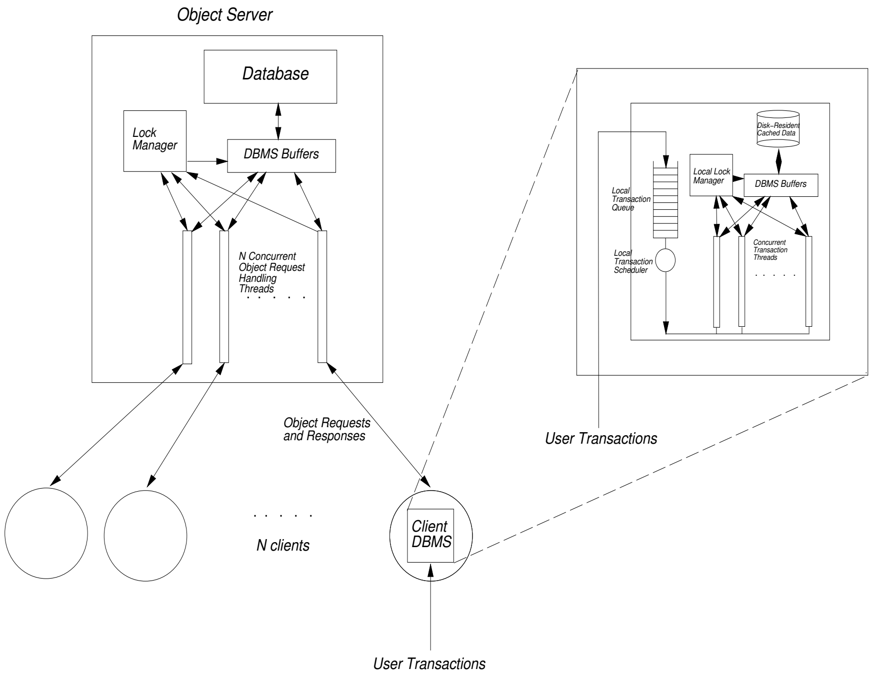

Figure 11: Implementation of the CS-RTDBS

The test environment for our experiments was a system consisting of five Sun ULTRA-1 workstations residing on an 10 Mbps Ethernet LAN. The database server executed by itself on one workstation and the clients were evenly distributed on the other four. We ran our experiments for two different database access workloads, a varying percentage of updates, and evaluated the four transaction scheduling policies. The duration of each experiment was approximately 2 hours, and therefore, a complete set of experiments took about 70 hours to perform. We ran each experiment several times in order to verify the confirm the consistency of the derived results.

Parameter	*Value*
Database Size	10,000
Centralized RTDBS Server Main Memory	5,000
CS-RTDBS Server Main Memory	1,000
Client Disk Cache Size	1,000
Client Memory Cache Size	600

Table 4: Prototype Parameters

The database consisted of 10,000 objects and the size of each object was 2 Kbytes. The database parameters for the prototypes is given in Table 4. In both systems, transaction arrival at each client is determined by a Poisson arrival process with a mean inter-arrival time of 10 seconds. The Average Transaction Length (ATL) –CPU processing time– is generated according to an exponential distribution with a specific average. The choice of the Poisson and Exponential distributions was made so as to be in accordance with theoretical queuing models. The values used for the ATL in our three sets of experiments are given in Table 5. Transaction deadlines were set according to an

Exponential distribution with thrice the average processing time of a transaction as its mean. The average number of objects accessed by each transaction is 10. The values for the average inter-arrival time and for the average CPU processing time of transactions (ATL) were chosen to be large enough to make the effects of network data transfer times negligible in comparison to those of real-time transaction scheduling and locking-related delays. The update workload is varied by using different update selectivities, i.e., the percentage of the overall object accesses that are modifications. We tested the two prototypes with 1% and 5% as well as a very extreme (for regular databases) 20% update selectivity.

We experimented with two different database access patterns that create different degrees of contention for database objects. In the first access pattern, we divided the database into ten equal-sized ranges. The probabilities of transactions' object accesses for each range were set according to the Normal distribution shown in Figure 12. Within each range, the actual objects to be accessed were selected with a Uniform probability. Hence, all clients demonstrate a very keen interest in a common "hot" area which is 20% of the database. The degree of contention for the objects in this hot area is expectedly very high. In the other access pattern, which we call *Localized-RW*, each client has its own distinct "hot" area that is 1% of the size of the database. 80% of a client's accesses are made to this hot area (according to the Uniform distribution) while the other 20% of the accesses were to the remainder of the database according to the Zipf distribution [20]. Therefore, Localized-RW is very similar to the HOTCOLD workload described in [7] except that a client's accesses to the rest of the database objects are not uniformly distributed. The Localized-RW pattern was designed to investigate the effect of locality in clients' data accesses on the efficiency of the CS-RTDBS. The Localized-RW database access scheme is depicted in Figure 13.

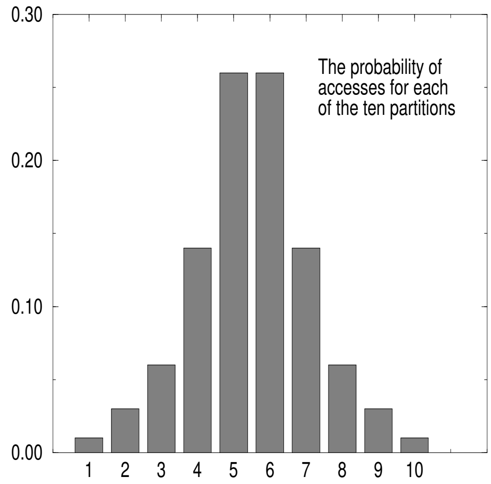

Figure 12: Normal Object Access Distribution

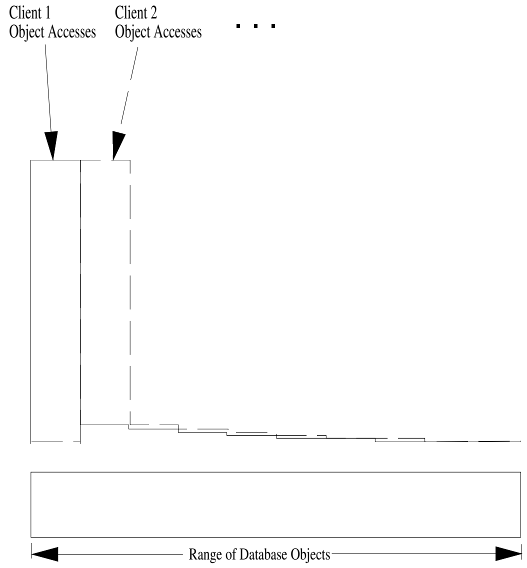

Figure 13: *Localized-RW* Object Access Distribution

In order to perform a comparison of the two processing models under varying workload conditions, we first tested them with the Normal and Localized-RW database access patterns for an average transaction processing time of 10 seconds. Then, we used the Localized-RW scheme and set the average transaction processing time to 20 seconds. The higher processing time per transaction allowed us to see the performance of the systems under overload conditions. The following subsections describes the results of our experiments for each of the above three workloads. For each experiment, our basic measure of system performance is the percentage of transactions that complete within their prescribed deadlines. Transactions that are aborted either during deadlock detection or because they have missed their deadlines are counted as failures.

Parameter	*Experiment Set 1*	*Experiment Set 2*	*Experiment Set 3*
Database Access Pattern	Localized-RW	Normal	Localized-RW
Average Transaction Inter-Arrival Time (seconds)	10	10	10
Average Transaction CPU Processing Time (seconds)	10	10	20
Average Number of Objects Accessed by Each Transaction	10	10	10
Update Selectivity	1%, 5%, 20%	1%, 5%, 20%	1%, 5%, 20%
Transaction Scheduling Strategies	ED,LS,FCFS,LAD	ED,LS,FCFS,LAD	ED,LS,FCFS,LAD

Table 5: Parameters for each set of Experiments

4.1 Localized-RW and Average Transaction Processing Time of 10 Seconds

The performance of the two real-time configurations, in terms of the percentage of transactions that completed within their deadlines, for an update selectivity of 1% is shown in Figure 14. The derived curves show that the CE-RTDBS offers greater efficiency for a very small number of clients. The faster processing ability and larger memory capacity of the centralized server outweighs the advantages offered by the CS-RTDBS at very low loads. However, as the load increases the performance of the centralized system deteriorates rapidly, independent of the scheduling algorithm used. This is because the distributed scheduling of the CS system tends to alleviate the problems caused by an unfavorable scheduling policy.

Additionally, the low percentage of conflicting lock requests makes the client caches in CS-RTDBS very efficient. Most object requests made by client transactions are satisfied locally. The low updates selectivity results in very few conflicting locks requests from clients. Therefore, object requests dispatched to the server are satisfied within very short times – see column for 1% update selectivity in Table 6. Requests for exclusive locks are satisfied in only slightly longer time than shared lock requests. All these factors cause the CS-RTDBS to demonstrate better performance than the CE-RTDBS as the number of clients increases beyond 40.

Figure 15 shows a break-up of the average object response times in the CS-RTDBS. Here, the response times for both shared and exclusive lock requests are shown as two components. The first component indicates the network transfer time for the object request from the client to the server plus the object transfer time in the reverse direction. The second component denotes the delay encountered in processing each object request (including the deadlock detection). A significant portion of this second component is the time required to call back an object from a client that has locked it in a conflicting mode. The actual overhead incurred in performing the deadlock detection itself is shown in Figure 16. What is shown here is the average time spent in deadlock detection per request including the time it takes to resolve deadlocks when they are detected. The time spent in checking whether a request could cause a cycle in the wait-for graph is very short for 20 clients (11 milliseconds), but when the number of clients increases to 100, this delay is as much as 74 milliseconds.

The transaction scheduling policies used have a noticeable effect on the performance of two configurations. In the CE-RTDBS, the LS scheduling policy outperforms FCFS by about 8%. In the CS-RTDBS, the difference between the best policy (LS) and the worst one (FCFS) is approximately 4%. It is interesting to note that the LAD scheduling policy performs better than the ED and FCFS methods. The *Localized-RW* access pattern allows a considerable percentage of client requests to be satisfied locally, and therefore, scheduling transactions on the basis of available local data is beneficial.

The percentage of transactions completed successfully by both systems for a 5% update selectivity is shown in Figure 17. The increased percentage of updates causes the efficiencies of both systems to drop by a small amount. In the CS-RTDBS, this is due to the more frequent transaction blocking at the clients for locks on objects, and an increased number of object callbacks and re-fetches. The increased data contention causes a reduction in the effectiveness of CE-RTDBS as well. However, since the data is all stored at the server, the only cost incurred

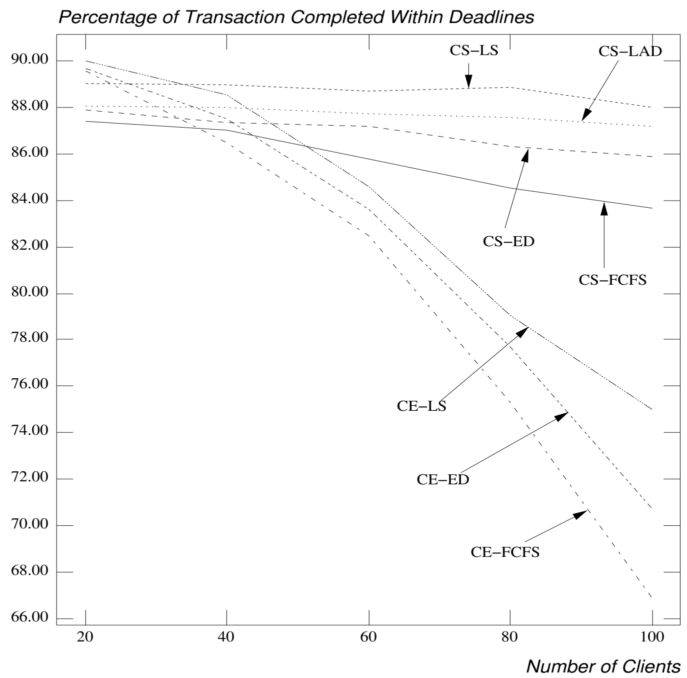

Figure 14: 1% updates (Localized-RW, ATL 10 seconds)

Number of Clients	*1% Update Selectivity*		*5% Update Selectivity*		*20% Update Selectivity*	
	Shared Lock	Exclusive Lock	Shared Lock	Exclusive Lock	Shared Lock	Exclusive Lock
20	43	58	57	90	102	109
60	53	181	97	185	226	242
100	58	251	135	257	363	382

Table 6: Average Object Response Times in the CS-RTDBS (in milliseconds) for the Localized-RW Access Pattern and ATL of 10 seconds (ED scheduling)

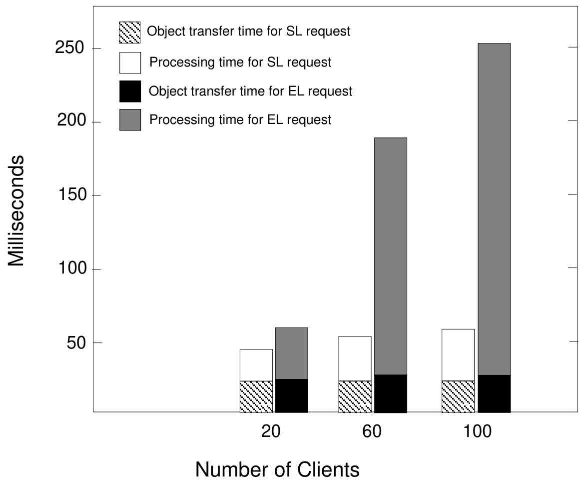

Figure 15: Breakup of Average Object Response Times in CS-RTDBS for (1% updates, Localized-RW, ATL 10 seconds, ED scheduling)

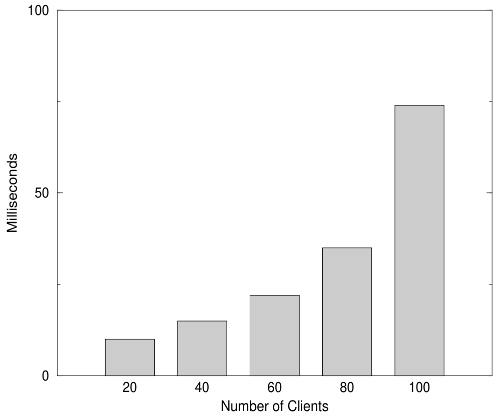

Figure 16: Deadlock Detection Overhead Per Object Request in CS-RTDBS (1% updates, Localized-RW, ATL 10 seconds)

is in the serializability of interleaved transactions requesting conflicting locks. Consequently, the centralized system performs better than the CS system for lower loads (up to 40 clients). Due to the rapid degradation of the performance of the CE-RTDBS, the CS-RTDBS performs better when the number of clients becomes large. Overall, the results for this update setting follow the same pattern observed for 1% updates although the benefits of the CS-RTDBS become apparent only for more than 60 clients. A similar pattern of results is repeated in the third group of experiments when the selectivity of updates is increased to 20%. Once the number of clients in the system becomes very high (greater than 60), the client-server system demonstrates greater efficiency. From Figure 19, it can be seen that the deadlock detection overhead (per request) is not higher for an update selectivity of 20% than that for 1% (Figure 16). This is because just increasing the proportion of exclusive lock requests does not cause an corresponding rise in the complexity of the wait-for graphs used to detect deadlocks.

An important observation that can be made from the results of the above three groups of experiments is that the performance differential between the two systems decreases as the percentage of updates increases. When the number of clients is large and the update selectivities are limited, the difference in the performance levels of the two systems is more notable. On the whole, the results of this set of experiments demonstrate that for reasonable percentages of updates the CS-RTDBS can significantly outperform its centralized equivalent.

4.2 Normal and Average Transaction Processing Time of 10 Seconds:

In this set of experiments, transactions at each client accessed the database according to the Normal distribution shown in Figure 12. For an update selectivity of 1%, the percentage of transactions that completed within their deadlines is depicted in Figure 20. The results follow a trend similar to that seen in the set of experiments with Localized-RW accesses. The CE-RTDBS performs very well for a small number of clients but its performance deteriorates very rapidly as the number of clients increases. The CS-RTDBS demonstrates stable performance

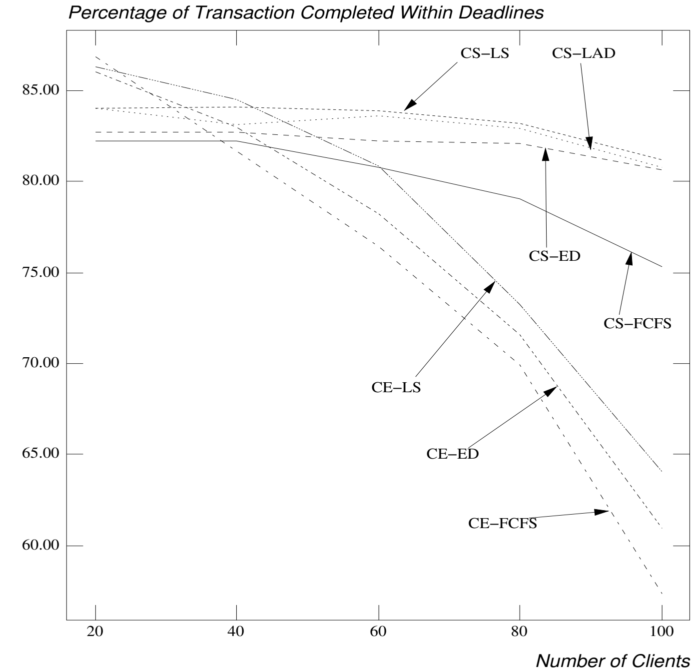

Figure 17: 5% updates (Localized-RW, ATL 10 seconds)

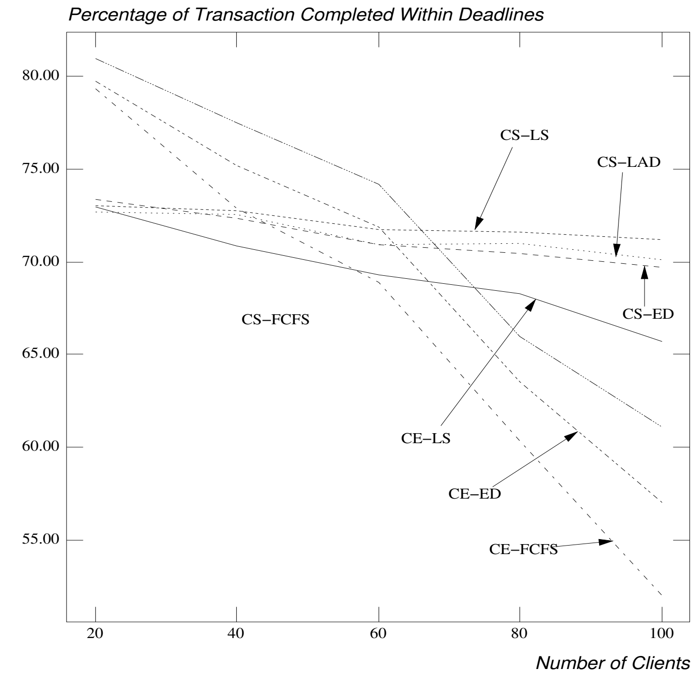

Figure 18: 20% updates (Localized-RW, ATL 10 seconds)

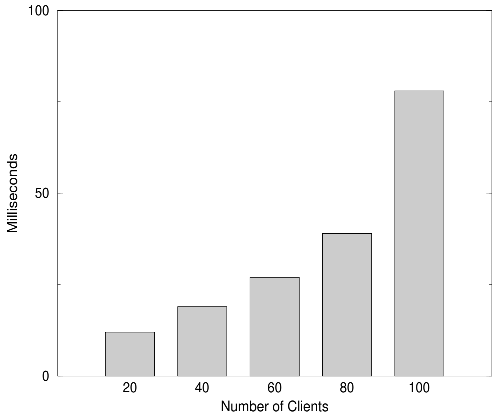

Figure 19: Deadlock Detection Overhead Per Object Request in CS-RTDBS (20% updates, Localized-RW, ATL 10 seconds)

levels as the number of clients increase. This is because the small percentage of updates allows quick servicing of clients' object requests (Table 7). Additionally, the very low probability of exclusive lock request conflicts allows clients to cache data for long periods of time. This allows transaction requests to be satisfied locally without reference to the server. Consequently, the percentage of the CS-RTDBS is better than that of the CE-RTDBS when the number of clients increases beyond 60. As Figure 20 shows, the scheduling policies used have a considerable impact on the efficiencies of the two configurations. Expectedly, LS demonstrates the best performance in either architecture, followed by ED and FCFS. In the CS-RTDBS, LAD does not perform any better than ED. This is because of the higher degree on contention for the more frequently accessed ranges of objects in the Normal distribution.

Number of Clients	*1% Update Selectivity*		*5% Update Selectivity*		*20% Update Selectivity*	
	Shared Lock	Exclusive Lock	Shared Lock	Exclusive Lock	Shared Lock	Exclusive Lock
20	58	265	66	264	94	279
60	63	538	88	562	201	724
100	69	850	90	917	355	1296

Table 7: Average Object Response Times in the CS-RTDBS (in milliseconds) for the Normal Access Pattern and ATL of 10 seconds (ED scheduling)

Increasing the selectivity of updates to 5% (Figure 22) adversely affects the performance of the two systems, but the overall trends similar to those in Figure 20. The higher percentage of object updates and the absence of locality in the clients' data accesses results in longer transaction blocking in the CS-RTDBS. In spite of these drawbacks, the CS-RTDBS is able to demonstrate greater efficiency when the number of clients is more than 80.

It is when the percentage of object updates is increased to a very extensive 20% that we can see a significant

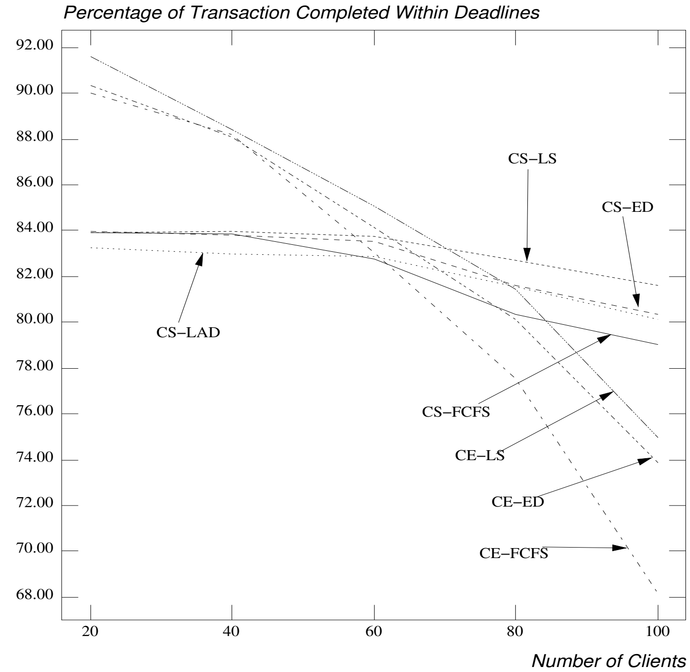

Figure 20: 1% updates (Normal, ATL 10 seconds)

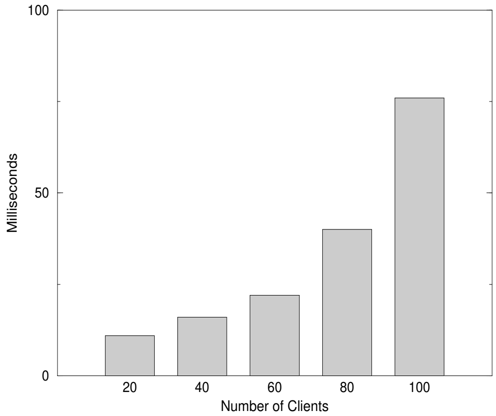

Figure 21: Deadlock Detection Overhead Per Object Request in CS-RTDBS (1% updates, Normal, ATL 10 seconds)

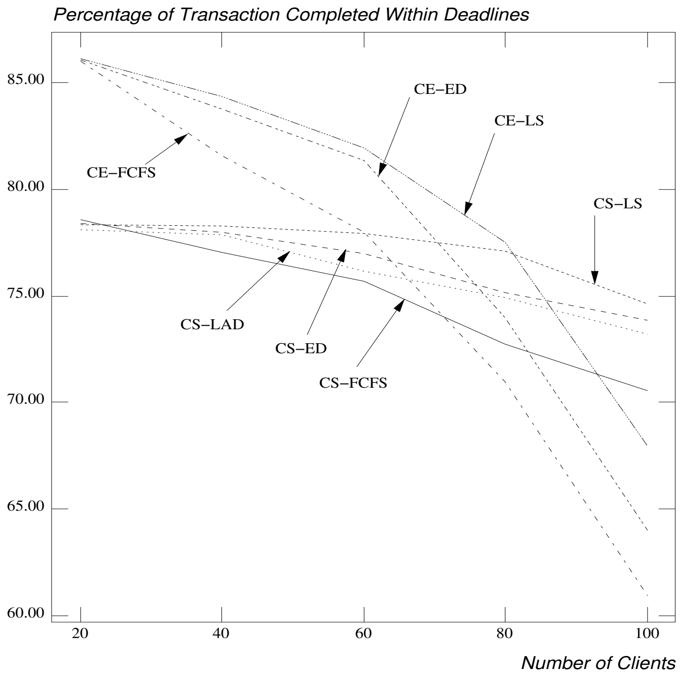

Figure 22: 5% updates (Normal, ATL 10 seconds)

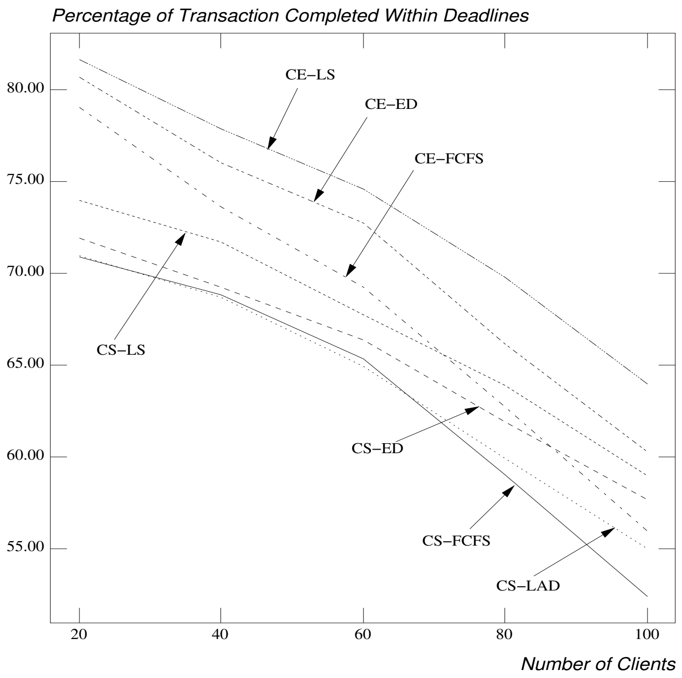

Figure 23: 20% updates (Normal, ATL 10 seconds)

deterioration in the performance of the CS-RTDBS. The dramatic increase in the percentage of updates causes a manifold increase in the percentage of conflicting lock requests. This implies that the clients have to return and re-fetch database objects to and from the server. The CE-RTDBS is affected by the high percentage of updates as well. However, since the data is all stored at one location, it does not incur the significant costs involved in transporting data back and forth between the clients and the server. The column for 20% updates selectivities in Table 7 shows the extensive delays in obtaining locks on database objects. For 100 clients, obtaining an exclusive lock takes 1.296 seconds on the average. This is a very considerable delay resulting in a worse performance for the CS-RTDBS than its centralized counterpart.

For the CS-RTDBS, the LAD scheduling policy is now noticeably worse than the LS and ED policies. Higher contention for objects in the clients' common hot area results in a very small percentage of these objects being readily available in local caches. Thus, the increased presence of non hot objects in each client's cache causes misjudgements in LAD's scheduling decisions.

Overall, the performance of the CE-RTDBS is slightly better for the Normal access distribution than the *Localized-RW* one. We believe that this is due to the fact that transactions' data accesses here are largely concentrated on a very small portion of the database (almost 52% of all object accesses are in ranges 5 and 6 of the database – Figure 12). The centralized server copes in a more effective way as frequently accessed objects remain in main memory buffers much more often, thus requiring a smaller number of disk accesses.

4.3 Localized-RW and Average Transaction Processing Time of 20 Seconds:

We, now, evaluate the systems when the average transaction processing time is 20 seconds. Since the mean transaction inter-arrival time is only 10 seconds, the systems are operating under significant overload conditions. The performance of the two systems for a 1% update selectivity can be seen in Figure 24. As expected, the CE-RTDBS behaves well when the number of clients is small. In fact, for 20 clients the CE-RTDBS completes almost 10% more transactions within their deadlines than its client-server counterpart. This is a very large margin in a real-time environment. However, this level of performance cannot be sustained by the CE-RTDBS as the load on the system increases. As the number of clients increases beyond 40, performance degrades very rapidly. For one hundred clients, the CE-RTDBS can complete only 58% transactions successfully with ED or LS transaction scheduling. With FCFS scheduling, this percentage is even lower at 46%. On the other hand, the client-server system demonstrates remarkably stable performance in the face of increasing transactional loads. For 20 clients the percentage of transactions that the CS-RTDBS can complete using the FCFS discipline is approximately 80%. This percentage declines very gradually to 74% when the number of clients attached to the server is increased to 100. The LAD scheduling policy is able to demonstrate performance equal to that of the ED policy. It is unable to do better than ED because the longer transaction execution times imply proportionally longer transaction blocking (in case of conflicting locks). In such situation, using the LS policy leads to higher efficiency.

We can see that the average deadlock detection overhead –shown in Figure 25– is higher than in the first two

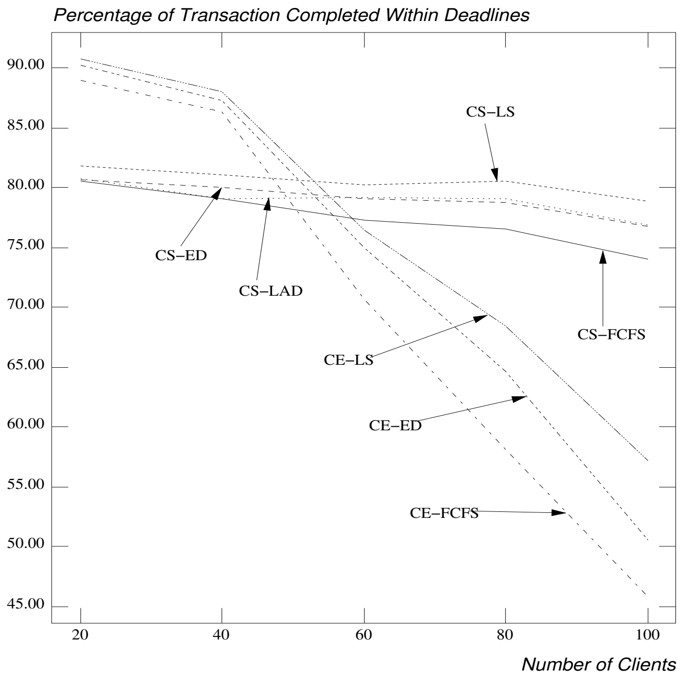

Figure 24: 1% updates (Localized-RW, ATL 20 seconds)

set of experiments. This is a direct result of the increase in the transaction CPU processing times. Locks once acquired by client transactions are not released for longer periods of time (twice as long on the average). This causes an increase in the number of outstanding requests in the wait-for graphs, and hence, increased delays in performing the deadlock detection. In the CS-RTDBS, for 100 clients, the average deadlock detection overhead is 101 milliseconds.

When the update selectivity is set to 5%, the results follow the same general trend as those in Figure 24 but absolute performance levels are slightly lower. The percentages of transactions that completed within their deadlines are shown in Figure 26. For a smaller number of clients (upto 40), the CE-RTDBS offers better performance but once the number of clients becomes high, the CS-RTDBS is clearly better. The increased contention for the database causes the performance of both systems to be affected adversely but the locality of accesses by the clients in the CS-RTDBS allows it to maintain a consistent performance level as the load increases. The effect of the increased percentage of updates on the CE-RTDBS becomes clearly visible when the load increases.

When the selectivity of updates is increased to 20%, we observe –as expected– that the performance of the CE-RTDBS is better than or equal to that of the CS-RTDBS independent of the number clients attached. The increased contention for exclusive locks causes the clients in the CS-RTDBS to have to return database objects to the server and re-fetch them much more often. The performance of the CE-RTDBS is also affected by the increased percentage of updates but since all the data is available in one location the overall overhead incurred by the system is smaller. The results of this set of experiments is shown in Figure 27.

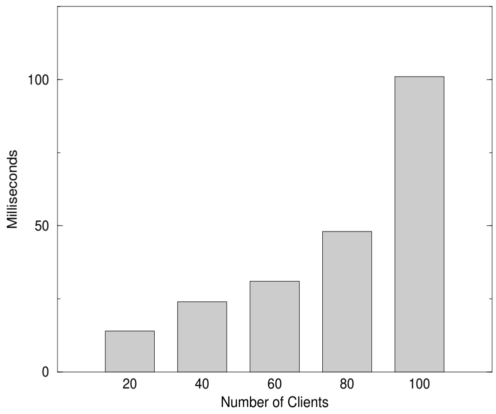

Figure 25: Deadlock Detection Overhead Per Object Request in CS-RTDBS (1% updates, Localized-RW, ATL 20 seconds)

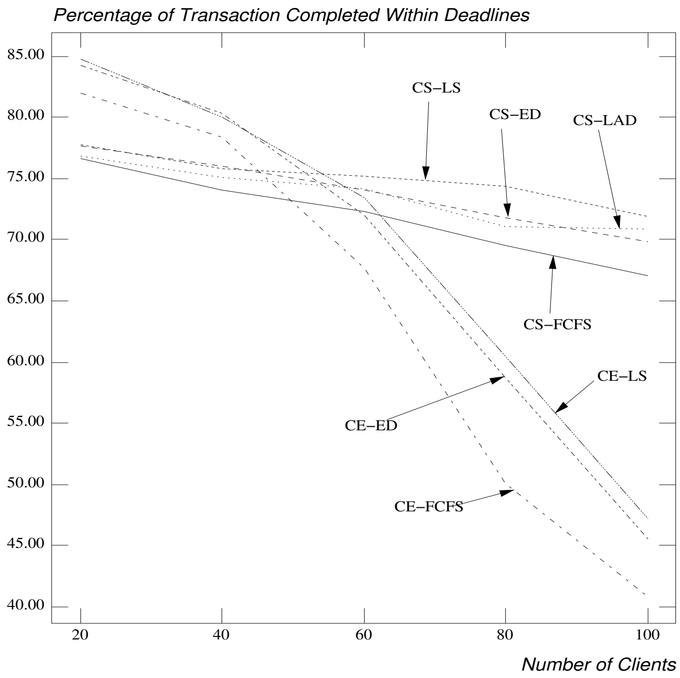

Figure 26: 5% updates (Localized-RW, ATL 20 seconds)

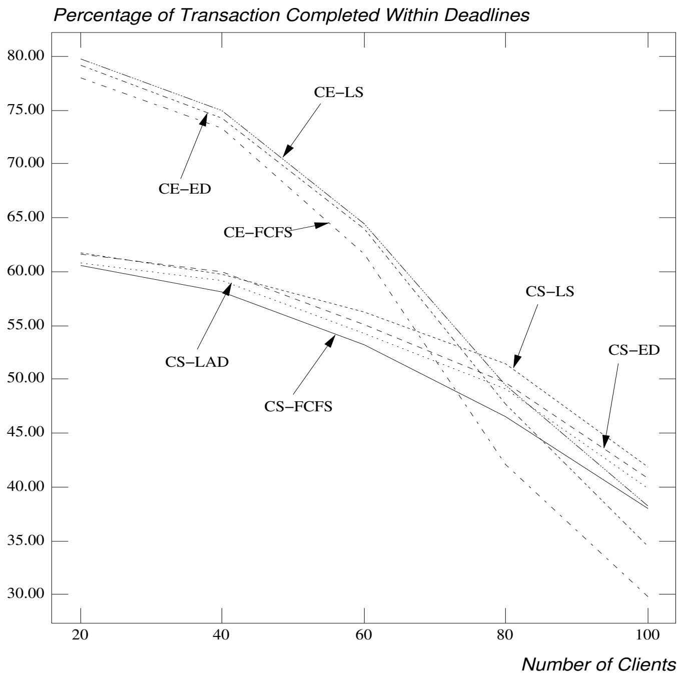

Figure 27: 20% updates (Localized-RW, ATL 20 seconds)

5 Related Work

There exists an extensive body of research in real-time systems indirectly related to our work. Here, we highlight a limited number of representative works that investigate issues in scheduling of real-time jobs, design issues in RTSs, and resource management in multiprocessor and distributed settings.

In [1], scheduling methods for database transactions are proposed and concurrency control policies are investigated through experimentation. The management of concurrent transactions in real-time settings is further discussed in [16, 19]. A family of speculative concurrency control techniques is proposed in [4]. Here, potentially costly delays due to blocking and roll-backs are avoided with the use of additional shadow transactions that guard incomplete database transactions. Haritsa et al. [17] propose the Adaptive Earliest Deadline scheduling algorithm (AED). To overcome the observed disadvantages of the Earliest Deadline First algorithm (ED) in a heavily-loaded RTDBS, AED features a feedback control mechanism that detects overload conditions and modifies transaction priority assignments accordingly. An alternative approach for real-time database systems is presented in [37]. Deadlines are associated with "contingency constraints" rather than directly with transactions.

The architectural design of StarBase is reported in [25]. StarBase is a firm RT-DBS based on the RT–Mach OS and the concept of precise serialization is used to reduce transaction aborts. The imprecise-computation model is augmented so that it accounts for input errors and real-time constraints in [12]. In this context, heuristic algorithms for the scheduling of preemptive and composite tasks are proposed and examined using a suite of simulation experiments. In [23], the fact that general complex distributed tasks consist of subtasks is exploited by the proposed scheduling algorithms. Here, it is assumed that such subtasks may be carried out in parallel.

The problem of scheduling tasks with deadlines and resource requirements is discussed in [31] in the context of two multiprocessor models (shared and local memory). In [6], the analysis of both on–line and off–line schemes for multiple processors running the rate–monotonic scheduling algorithm is presented. Assignment of priorities in $active$ real-time multiprocessor environments is discussed in [36]. Three policies that take into account the amount of active work generated by transactions are proposed and their behavior is investigated with simulation experiments.

In contrast to the above efforts, our approach not only advocates usage of the CS model but also makes opportunistic use of client resources. It is this type of distributed processing that, in many instances, can produce higher completion rates for real-time transactions in comparison to centralized systems.

6 Conclusions

As network-centric computing becomes commonplace in today's business environments, client-server platforms handling time-constrained requests emerge as the dominant framework for transaction processing. In this paper, we propose the usage of the client-server database paradigm for handling requests with time constraints. The resulting configuration takes advantage of the available client resources and schedules requests locally. We have established the viability of the Client-Server Real-Time Database (CS-RTDBS) configuration by contrasting its performance with a comparable centralized system (CE-RTDBS) in a number of diverse workload settings. For our experimental study, we have developed prototype packages for both CE-RTDBS and CS-RTDBS using the Solaris 2.5 thread library, TCP sockets, and a database system library which allows paged I/O.

The results of our experimental effort indicate that:

- For workloads with usual to large database update selectivities (i.e., 1% and 5% of all data access modify data), the CE-RTDBS offers better performance than the CS-RTDBS when the number of clients is small. However, as the number of clients attached to the server is increased, the CS-RTDBS is consistently more efficient. For all three workloads that we examined, the performance of the centralized system degrades very rapidly as the number of clients increased.

- When the percentage of updates is significantly increased to 20%, the CE-RTDBS performs better than or equal to the CS-RTDBS for the Normal database access pattern and for the overload situation with the Localized-RW access scheme.

- The scheduling strategy used to assign priorities to transactions affects the performance of the CE-RTDBS (in terms of percentage of transactions completed within their deadlines) much more significantly than that of the CS-RTDBS. In the CS-RTDBS, the transaction scheduling strategy based on the local availability of data (LAD) performs better than the ED or FCFS policies when the database clients demonstrate locality in their object accesses. The LS policy performs better consistently. However, in the absence of accurate transaction execution times, the LAD discipline does provide a viable alternative for effective scheduling at the client sites.

- The time spent by transactions awaiting server data is seen as a very significant factor affecting the performance of the CS-RTDBS. Thus, the minimization of server-related delays, by using inter-transaction caching of data objects/locks, and allowing independent transaction scheduling at the clients serve as the enablers for higher CS transaction completion rates.

Based on the results of the experimentation, we believe that the client-server paradigm is an effective and scalable alternative to the centralized model for real-time transaction processing.

We plan to extend our work while pursuing at least three different directions: (i) techniques that enhance the basic data-shipping model and improve its real-time processing capabilities; the adaptive merging of data and query-shipping based on the location of data within the system is such a method, (ii) distributed speculative transaction processing where the use of additional system resources can provide lock conflict resolution and minimize the occurrences of transaction rollback and blocking, and (iii) logical clustering of database clients based on their data access patterns – this can significantly reduce the load on the object server resulting in reduced transaction blocking.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3), 1992.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational Transducers for Electronic Commerce. In *Proceedings of ACM Symposium on Principles of Database Systems*, Seattle, WA, June 1998.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of ACM-SIGMETRICS Conference*, Santa Clara, CA, June 2000.

[4] A. Bestavros and S. Braoudakis. Timeliness via Speculation for Real–Time Databases. In *Proceedings of the IEEE Real–Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[5] W. Bolosky, J. Douceaur, D. Ely, and M. Theimer. Feasibility of Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of ACM-SIGMETRICS Conference*, Santa Clara, CA, June 2000.

[6] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. Assigning Real–Time Tasks to Homogeneous Multiprocessor Systems. *IEEE Transactions on Computers*, 44(12), December 1995.

[7] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client–Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD Conference*, Denver, CO, May 1991.

[8] I. Chu and M. Winslett. Choices in Database Workstation-Server Architecture. In *Proceedings of the 17th Annual International Computer Software and Applications Conference*, Phoenix, AZ, November 1993.

[9] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. *IEEE Transactions on Software Engineering*, 19(2):120–138, February 1993.

[10] D. DeWitt, D. Maier, P. Futtersack, and F. Velez. A Study of Three Alternative Workstation–Server Architectures for Object–Oriented Database Systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 107–121, 1990.

[11] A. Dogac. A Survey of the Current State-of-the-Art in Electronic Commerce and Research Issues in Enabling Technologies. In *Proceedings of the Euro-Med Net 98 Conference, Electronic Commerce Track*, March 1998.

[12] W. Feng and J. Liu. Algorithms for Scheduling Real-Time Tasks with Input Error and End-To-End Deadlines. *IEEE Transactions on Software Engineering*, 23(2), 1997.

[13] WAP Forum. Wireless Application Protocol. http://www.wapforum.org, June 2000.

[14] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

[15] A. Gal, S. Kerr, and J. Mylopoulos. Information Services on the Web: Building and Maintaining Domain Models. *International Journal of Cooperative Information Systems*, 8(4):227–254, 1999.

[16] J. Haritsa, M. Livny, and M. Carey. On Being Optimistic about Real–Time Constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990.

[17] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proceedings of the 12th Real-Time Systems Symposium*, San Antonio, TX, USA, December 1991.

[18] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[19] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Concurrency Control Schemes. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

[20] Y. Ioannidis. Universality of Serial Histograms. In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[21] E. Jensen, C. Locke, and H. Tokuda. A Time-Driven Scheduler for Real–Time Operating Systems. In *Proceedings of the IEEE Real–Time Systems Symposium*, pages 112–122. IEEE, 1985.

[22] V. Kanitkar and A. Delis. A Case for Real-Time Client-Server Databases. In *Proceedings of the 2nd International Workshop on Real-Time Databases*, Burlington, VT, USA, September 1997.

[23] B. Kao and H. Garcia-Molina. Subtask Deadline Assignment for Complex Distributed Soft Real–Time Tasks. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[24] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10), October 1991.

[25] M. Lehr, Y. Kim, and S. Son. Managing Contention and Timing Constraints in a Real–Time Database System. In *Procs. of the 16th IEEE Real–Time Systems Symposium*, Pisa, Italy, December 1995.

[26] C. Mohan and I. Narang. Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment. In *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 453–468, Vienna, Austria, March 1992.

[27] C. Mohan and I. Narang. ARIES/CSA: A Method for Database Recovery in Client–Server Architectures. In *Proceedings of the 1994 ACM SIGMOD Conference*, pages 55–66, Minneapolis, Minnesota, May 1994.

[28] A. Mok and M. Dertouzos. Multiprocessor Scheduling in a Hard Real–Time Environment. In *Proceedings of the 7th Texas Conference on Computing Systems*, 1979.

[29] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Client-Based Logging for High Performance Distributed Architectures. In *Proceedings of the 12th International Conference on Data Engineering*, pages 344–351, New Orleans, LA, USA, Feb-March 1996.

[30] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, Paris, France, 1990.

[31] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[32] B. Reich and I. Ben-Shaul. A Componentized Architecture for Dynamic Electronic Markets. *ACM SIGMOD Record*, 27(4), December 1998.

[33] N. Roussopoulos and A. Delis. Modern Client–Server DBMS Architectures. *ACM SIGMOD Record*, 20(3):52–61, September 1991.

[34] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of $ADMS\pm$. *IEEE Computer*, 19(2):19–25, December 1986.

[35] J. Santos, R. Muntz, and B. Ribeiro-Neto. Comparing Random Data Allocation and Data Stripping in Multimedia Servers. In *Proceedings of ACM-SIGMETRICS Conference*, Santa Clara, CA, June 2000.

[36] R. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham. Priority Assignment in Real-Time Active Databases. *The VLDB Journal*, 5, 1996.

[37] N. Soparkar, H. Korth, and A. Silberschatz. Databases with Deadline and Contingency Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), August 1995.

[38] W. Stallings and R. Van Slyke. *Business Data Communications*, chapter Wireless Networks. Prenctice Hall, Upper Saddle River, NJ, 2001.

[39] J. Stankovic. The Many Faces of Multi-Level Real–Time Scheduling. In *Proceedings of Real–Time Systems and Applications*, pages 2–5, Tokyo, Japan, October 1995.

[40] MiniRel Development Team. The MiniRel Relational DBMS. University of Wisconsin, Madison, 1989.

[41] A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[42] O. Torbjornsen and S. Bratsberg. Designing an Ultra Highly Available DBMS. In *Proceedings of ACM-SIGMOD Conference*, Dallas, TX, May 2000.

[43] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. In *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989.

[44] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD Conference*, Denver, CO, May 1991.

[45] K. Wilkinson and M. Neimat. Maintaining Consistency of Client-Cached Data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 122–133, Brisbane, Australia, August 1990.