Distributed and Parallel Databases, 17, 39–74, 2005

© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Efficient Processing of Client Transactions in Real-Time*

VINAY KANITKAR kanitkar@akamai.com
Akamai Technologies Inc., 8 Cambridge Center, Cambridge, MA 02139, USA

ALEX DELIS ad@naxos.poly.edu
Department of CIS, Polytechnic University, Brooklyn, NY 11201, USA

Recommended by: Athman Bouguettaya

Abstract. In traditional client-server databases, a transaction and its requisite data have to be colocated at a single site for the operation to proceed. This has usually been achieved by moving either the data or the transaction. However, the availability of high-bandwidth networking options has led users of today's systems to expect real-time guarantees about the completion time of their tasks. In order to offer such guarantees in a client-server database system, a transaction should be processed by any means that allows it to meet its deadline. To this end, we explore the option of moving both transactions and data to the most promising sites for successful completion. We propose a load-sharing framework that oversees the shipment of data and transactions so as to increase the efficiency of a cluster consisting of a server and a number of clients. Here, efficiency is defined as the percentage of transactions successfully completed within their deadlines by the cluster. The suitability of a client for processing a transaction is measured with respect to the availability of the transaction's required data in its local cache. In addition to the load-sharing algorithm, we use the concept of grouped locks, along with transaction deadline information, in order to schedule the movement of data objects in the cluster in a more efficient manner. We evaluate the real-time processing performance of the client-server architecture using detailed experimental testbeds. Our evaluation indicates that it is possible, in many situations, to achieve better performance than a centralized system.

Keywords: client-server databases, real-time transaction scheduling, transaction-shipping

1. Introduction

Implementations of contemporary database systems have often been based on the client-server framework. These systems are expected to provide distributed access to shared data and also support real-time constraints on individual tasks in order to support applications in financial environments, process control systems, and computer integrated manufacturing. Client-server databases (CSD) have utilized the processing capabilities and network bandwidths available today in order to successfully manage data and provide high transaction throughput. However, transaction processing in a CSD in the presence of deadlines has not

*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.

been examined in much detail. We believe that this is an important new area of research as deployments of CSDs over local area networks and the world-wide web proliferate.

Transaction processing in a CSD has generally been performed either by sending the transaction to the location of the data (transaction-shipping) [16, 27], or by moving the data to the location of the transaction (data-shipping) [12, 34, 54, 55]. Although both these techniques offer several advantages, individually they have limited flexibility for use in a real-time application environment. In this paper, we propose a new framework that ships the data or the transaction or both to the site that is most likely to execute the transaction within its deadline and, hence, increases the efficiency of the system. Deadlines in real-time systems are usually introduced to specify quality of service requirements or control the operation of physical systems [37]. Our proposed framework can be used to facilitate the effective development of a wide range of application systems. For example:

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 [31, 53]. 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 only be 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 [22, 41].
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 [3, 48]. These facilities could be implemented as clusters of sites with characteristics similar to those advocated in this paper.
4. *Distributed dynamic content assembly*: With the advent of Internet banking and financial services, web-site content has to be customized specific to each user. For example, a web-based stock trading service will allow its customers to configure their start pages so as to reflect their specific investment interests. A client-server cluster, as described in this paper, can be used to implement an application that allows such customized pages to be dynamically constructed at any client to which the request gets directed [3, 23].
5. *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 [2, 4, 9, 11, 19]. In doing so, organizations can overcome overload and response delays. The development of such multi-server farms can follow our system model.

In a hard real-time system, an operation is considered successful only if it finishes its execution within its specified deadline. Therefore, the key measure of system performance

is the percentage of all operations that complete within their deadlines. We use the same metric for measuring the effectiveness of our proposed techniques. This is in contrast with traditional database systems where performance is measured in terms of the transaction throughput or the average transaction turnaround time. One important distinction that we make from true hard real-time systems is that transactions that miss their deadlines are not aborted. Their deadlines are simply reset to infinity. This allows the system to continue to make progress on late transactions, but only when no transactions with earlier deadlines are available for consideration. In our previous work [24, 25], we have shown that a client-server database system can be more efficient than a centralized system (for real-time processing) in the presence of the following conditions: (i) if there is a reasonable amount of spatial and temporal locality in client data access patterns, and (ii) the percentage of data modifications is low. However, the presence of a high percentage of updates can cause centralized systems to perform better than their client-server counterparts. This is because of the increased overhead that client-server systems incur in ensuring distributed data consistency and in shipping data between sites as required. And, transactions at client sites are forced to block for long periods of time waiting for their required data objects and locks to become available.

In order to avoid these long periods of blocking, we propose a load-sharing algorithm that can significantly improve the efficiency of a client-server real-time database system (CS-RTDBS). Superimposed on a traditional data-shipping environment, our load-sharing algorithm heuristically selects the client site that is most likely to complete a given transaction within its deadline. The suitability of each site is evaluated by using heuristics that represent the availability of data and the current processing load at that site. In addition to the load-sharing algorithm, we use three additional techniques in order to improve the performance of the client-server architecture, namely: (i) transaction decomposition: splitting transactions into sub-tasks that can be executed independently, (ii) object request scheduling: using transaction deadline information to schedule the order in which object requests from clients are satisfied, and (iii) object migration planning: grouping requests for the same data in order to plan the movement of that data among the client sites in the system. We believe that the use of the load-sharing algorithm along with the above three enhancements will allow the CSD to demonstrate a better level of performance than comparable centralized processing environments. In order to verify the validity of our hypotheses, we have developed detailed prototypes of the centralized (CE-RTDBS), basic data-shipping client-server (CS-RTDBS), and the load-sharing client-server configurations (LS-CS-RTDBS). The results of our experiments show that the basic CS-RTDBS can indeed provide significant performance gains over the centralized architecture. Using the load-sharing heuristics improves the real-time processing performance of the client-server architecture (LS-CS-RTDBS) even further. This is because the primary weakness of the CS-RTDBS lies in the fact that client transactions have to block as they wait for data objects to be sent to them. The load-sharing technique that we employ in the LS-CS-RTDBS avoids such blocking and allows transactions to migrate to the client sites that hold the required data objects (along with the appropriate locks). This contributes significantly towards reducing the number of transactions that miss their deadlines.

Although client sites in this paper seem to be directly accessible by end-users, that need not be the case. The proposed computing cluster can be visualized as the "nucleus" system configuration running applications on a dedicated or virtual private network (VPN). Database clients are therefore trusted entities belonging to the coreapplication environment. The transfer of raw data objects and the execution of transactions is restricted to these machines only. End users of the system and their access points reside outside this trusted system and interact with the clients in order to submit their transactions and receive the results of their operations. Appropriate client sites are chosen either on the basis of their physical proximity to users or due to load balancing and data sharing considerations.

The rest of the paper is organized as follows. The next section discusses the operation of the data-shipping client-server database model. In Section 3, we describe the transaction-shipping technique and outline our load-sharing algorithm. Section 4 describes additional techniques of transaction decomposition, object request scheduling, and object migration planning. Testbed creation is described in Section 5 and the experimental results are presented in Section 6. Section 7 describes related work and the last section presents our conclusions.

2. The client-server database model

In contemporary data-shipping CSD architectures, clients have not only significant processing and main memory capabilities but also considerable persistent storage [12, 17, 28]. In the following discussion, we assume that the database is a collection of uniquely identifiable objects [13]. Users of the system initiate their transactions at client sites. The client sites are assumed to be homogenous to the extent that any client site can process all transactions made to the CSD. If the data objects necessary for a transaction's processing is not available at the client then it is requested from the server. The server responds by shipping the requested objects to the client. Once the client has acquired the appropriate objects, the local buffer space (disk and main memory) and CPU of a client are used to execute the transaction. The set of objects cached at a client is treated as a local data-space and is stored in the client's short and long-term memory (figure 1). Clients are permitted to hold on to data objects even after the transaction that requested them has completed. Future requests on this cached data can now be satisfied at the client itself without having to reference the database server.

As several clients can cache a particular database object, a system-wide concurrency control mechanism is required to serialize accesses to client-cached (and replicated) data. In this concurrency control mechanism, two kinds of locks are permitted: *shared* (read only) and *exclusive* (read-write). The server maintains a global lock table where, for each object, it stores the lock that has been granted to each client. A client transaction can update a cached database object only if the client has acquired an exclusive lock on that object. Unserialized accesses to the database are prevented by ensuring that no two clients are able to acquire conflicting locks on an object simultaneously. Therefore, an object can be locked by a client only after all conflicting locks on that object have been released. If a client's lock request on an object conflicts with the locks granted to other client(s) then the server sends callback messages to all such client(s) requesting that they release their locks as soon as

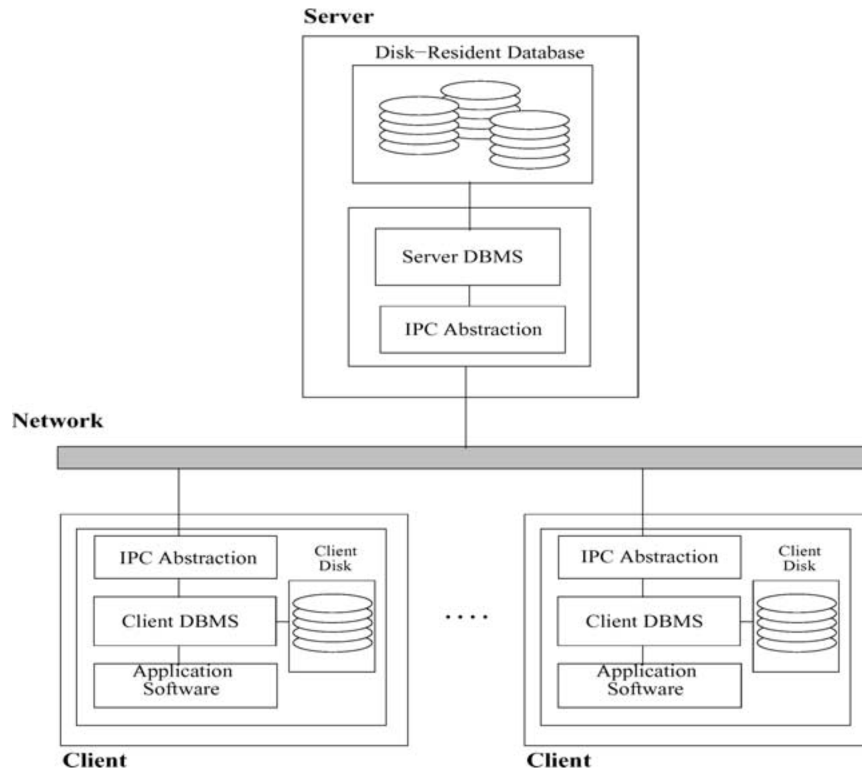

Figure 1. The client-server architecture.

possible. Only when a client has updated an object does it have to ship the new version of the object back to the server. Once all the conflicting locks have been released, the server grants the lock to the requesting client and sends the object over. 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.

The above locking scheme is pessimistic in nature and is meant to guarantee that accesses to the database are serializable. However, it can cause unnecessary blocking of transactions as they wait for the requested locks to be granted. To reduce such blocking delays, we have modified the above locking mechanism as follows. When the server requests a client to give up an exclusive lock on an object, it also specifies the type of lock the newly requesting client desires. If the newer request is for a shared lock then the client that holds the exclusive lock waits until its local transactions no longer need that exclusive lock and then returns the object to the server, at the same time, *downgrading* its own lock to a shared lock. Now, transactions at both clients can continue to access the object in shared mode. This novel technique is expected to improve data sharing among multiple sites, especially in low update situations.

Each client is allowed to execute multiple transactions concurrently.Therefore, concurrent transactions have to acquire appropriate locks on the objects they access. To manage such client-side locking, each client has a local lock manager. Transactions may request shared or exclusive locks on locally cached objects but, the lock that can actually be granted by the local lock manager depends on the lock that the client has acquired from the server—only a lock with lower access privileges can be granted. Therefore, a client transaction cannot get an exclusive lock on a cached object if the client itself has only acquired a shared lock from the server. This strict, hierarchical locking policy ensures that conflicting locks cannot be simultaneously granted to client transactions anywhere in the system. 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. As transactions execute entirely at one client site, deadlock detection at the clients can be performed independent of the global deadlock detection performed by the server.

In this paper, we evaluate real-time transaction processing in the abovedescribed CSD architecture under various workload settings. We also investigate the effectiveness of our load-sharing algorithm and the other three enhancement techniques. In the real-time context, we term the CSD as a Client-Server Real-Time Database System (CS-RTDBS). The basis of our evaluation is a comparison of this CS-RTDBS with a baseline Centralized Real-Time Database System (CE-RTDBS). Our measure of efficiency is the percentage of transactions completed within their deadlines. In the CE-RTDBS, the database server processes all the submitted transactions. Clients are assumed to be simple terminals and serve as user-interface points only. Clusters of terminals (clients) are managed by terminal servers that handle all communication between the clients and the centralized server. Transactions are initiated at the clients and are immediately shipped to the server for execution. The server schedules and executes these transactions according to the priority assignment algorithm in use. Two-phase locking is used to ensure that all accesses to database objects are serializable. The results of executing the transactions are communicated to users through their terminals. This is an early form of the query-shipping approach [39].

In both architectures, we assign priorities to real-time transactions using two very distinct policies. The first policy, called *Earliest Deadline First* (ED), performs its scheduling solely on the basis of the transactions' deadline information [1]. ED assumes that the system has no knowledge of task CPU execution times and assigns the highest priority to the transaction with the earliest deadline. Since ED does not use any information about the (estimated or exact) processing time of jobs it can schedule jobs that have missed their deadline. The second scheduling discipline presumes that the exact CPU processing time of each task is known. The task assigned the highest priority is the one with the least available slack—calculated as the difference between the deadline of a task and its required processing time. This is, therefore, known as *Least Slack First* (LS) scheduling. In both scheduling policies, we ensure that tasks that have missed their deadlines are not processed at all. Furthermore, in the LS algorithm, we also abort transactions that are expected to miss their deadlines. This is a straightforward task as the processing time of each task is known accurately. We have not considered the *First-Come First-Serve* (FCFS) scheduling policy.

This is because FCFS prioritizes tasks only on the basis of their arrival times, and is therefore, not well suited for real-time scheduling. Finally, it is important to note that in a database system, the availability of the appropriate data objects and locks is crucial to the timely execution of any transaction. In some cases, it is possible that a higher priority transaction may be executed after one with a lower priority as the latter has its requisite data objects readily available. This type of *priority inversion* is unavoidable in real-time systems, especially distributed ones [57].

In our experimental evaluation, we first compare the performance of the abovedescribed CS-RTDBS architecture with its centralized counterpart. Then, we evaluate the impact of using our load-sharing algorithm. This algorithm has been specifically designed for the CS-RTDBS and it uses information about the location of the data and the clients' processing loads to select the client site where a transaction could be processed more efficiently.

3. Load sharing

In this section, we describe our load-sharing algorithm that employs transaction-shipping in a data-shipping client-server database system in order to minimize the number of transactions that miss their deadlines. We first discuss the transaction-shipping mechanism and the advantages it offers. In the second subsection, we describe the heuristics that we use to decide when transaction-shipping could be beneficial. The load-sharing algorithm itself is presented in the third subsection.

3.1. Transaction-shipping

In data-shipping CSDs, clients are permitted to cache data objects indefinitely and also maintain locks on them. At times, a transaction at a client may not be able to commence its execution as other sites may have locked solicited objects in a conflicting manner. In a traditional data-shipping environment, the transaction is required to block until the server is able to recall all such conflicting locks. Instead, in some cases, it may be beneficial for clients to ship such transactions to sites that have locked (and cached) the object(s) in question. This type of task diversion requires up-to-date knowledge of database object/lock locations as well as current client processing loads. The server can provide this information to the clients as it maintains the lock table for all database objects in order to co-ordinate global concurrency control. There are specific two cases when it is advantageous to make use of such a transaction-shipping mechanism:

1. When a greater percentage of a transaction's required data is already cached at another site. For instance, consider a transaction initiated at $Client_A$ that requires access to four objects. Only one of these four objects is available at $Client_A$ while the rest are cached at $Client_B$. In this situation, it is almost certain that the transaction could finish earlier if it were transported to $Client_B$ for processing along with the one object available at $Client_A$.

2. When the client where the transaction was launched is heavily loaded. The load is often depicted as CPU utilization [32], but in contemporary client machines aggregates of resource metrics need to be considered such as utilization of buffer space, disk unit, network interface as well as CPU utilization.

Transaction-shipping offers a number of advantages in settings with real-time characteristics, namely: (i) clients can devote more resources to transactions that can be immediately executed locally, (ii) a transaction that has been shipped to another site will have at least as much chance of successful completion at that site as at its originating site, and (iii) network traffic can be reduced since the transfer of transactions may require less bandwidth than transferring database objects.

We use transaction-shipping in our load-sharing algorithm for processing real-time transactions so long as there is sufficient scope for gains in performance. To allow us to decide whether a transaction should be migrated to another site than its originating one, we have developed two heuristics that take into consideration the load on candidate clients as well as the availability of data at each of them. These heuristics are described in the next subsection.

3.2. Heuristics for load sharing

An important advantage that CSDs demonstrate is that the clients and the server are in frequent contact. Therefore, information about the current processing load at clients can be conveyed to the server by piggybacking object requests and/or releases. In this manner, the server can always maintain up-to-date information about the load on each client without incurring additional messaging and time overheads. This load-related information, consisting of the transaction queue lengths and buffer-space availability, can be used in conjunction with transaction-shipping to delegate transactions to lightly loaded clients. However, unlike centralized or shared memory systems, executing a database transaction in a distributed environment requires the physical availability of specific data objects and appropriate locks on them. These objects/locks have to be acquired from the database server or other clients. Hence the availability of the appropriate data plays an important role in making load-sharing decisions.

To use transaction-shipping for load sharing, there are two decisions to be made: firstly, whether a transaction should be shipped elsewhere for processing, and secondly, which site to ship it to [18, 58]. As described earlier in this section, there are two reasons to ship a transaction to another site. First, if a transaction is expected to miss its deadline at the client where it is initiated then, it is best to find a lightly loaded client to delegate this transaction to. This is estimated using the heuristic H_1. The second factor to be considered in deciding whether to ship a transaction to another site is whether the transaction could begin execution at that site with a shorter period of time spent in blocking. Our previous experiments with CS-RTDBSs have shown that the transport of objects to client sites does not contribute significantly to this blocking period [24]. In that setting, where the database is large collection of fairly small objects, most of the time spent in obtaining objects/locks is in waiting for clients that have conflicting locks to release them. Therefore, executing a

transaction at a site that needs to acquire the least number of conflicting locks from other sites is the most efficient available option. This is described below as the heuristic H_2. In the following description of the two heuristics, consider a transaction T that has been initiated at $Client_A$.

- H_1: We calculate this heuristic under two assumptions:
  (i) *If the CPU processing time of each transaction is not known*: if $Client_A$ has n transactions before transaction T in its priority queue then T is judged to have a *reasonable chance* of successfully completing at $Client_A$ if

$$(CurrentTime + n * ATL_A) \leq T_{\text{deadline}} \quad (1)$$

  where ATL_A is the average execution time for all completed transactions at $Client_A$.
  (ii) *If the CPU execution time of each transaction is known precisely*: Whenever a new transaction is initiated, the queue of waiting transactions is prioritized according to the *Least Slack First* (LS) policy—transactions that are already being considered for shipping are not considered here. The transactions are now examined in their LS order to check whether each of them can meet its deadline. Any transaction that cannot meet its deadline is considered for shipment to another site.
  If a transaction can reasonably be expected to complete at the client, then the available buffer space is examined to ensure that it is sufficient to accommodate the new task in addition to the current ones.
- H_2: The time spent by a transaction waiting for its requisite data to become available is the primary factor responsible for the failure of real-time transactions [24]. Most of this blocking time is spent in waiting for clients with conflicting locks to release them. Therefore, for Transaction T, $Client_A$ is considered to be a better processing site than $Client_B$ if T has to wait for fewer conflicting locks to be released if it were to be processed at A.
  Additionally, even if a transaction at $Client_A$ has fewer conflicting lock releases to wait for at another client, it is only shipped to that client if this does not require $Client_A$ to release any exclusive locks that it presently holds.

Our load-sharing algorithm uses the above two heuristics to gauge when a transaction could be executed more efficiently at another client site. This algorithm is presented in the next subsection.

3.3. The load sharing algorithm

In the description of the load-sharing protocol, we use granting of locks and the shipping of objects to mean the same thing. The load-sharing algorithm works as follows:

BEGIN

- Transaction T is initiated at a client.
- **IF** T can be accommodated in the local processing queue with a *reasonable chance* (H_1) of meeting its deadline **THEN**
  BEGIN
  - the client looks in its local cache for the objects/locks requested by the transaction. For all objects that are not available locally the client sends objects requests to the server. The current load at the client and available buffer space are piggy-backed on these object requests. This information is used by the server to update its load tables.
  - **IF** the server can grant all the objects/locks requested by transaction T **THEN**
    BEGIN
    these objects are locked appropriately on behalf of that client and shipped to it. The transaction is executed by the client locally.
    END;
    ELSE BEGIN
    * the server does not ship any objects over but instead sends the locations of the objects requested (by the transaction) that have been locked in conflicting modes by other clients. It also sends the current load estimate for these clients.
    * **IF** another client is in a better position to complete this transaction (H_2) **THEN** the transaction is shipped to it. The results of executing the transaction are communicated to the originating client if necessary.
      ELSE the client sends a message to the server indicating that the transaction will be processed locally and asks that the requisite objects be shipped over as soon as possible.

    END;

  END;
  ELSE BEGIN
  - the client queries the server for information about the location of the required objects and the processing loads on the other clients.
  - Once this information is received, the most suitable client (H_1 and H_2) is picked and the transaction is shipped to that client. If no client is more favorable according to H_2, then the transaction is shipped to the least loaded client according to H_1. Requests for objects required by that transaction are sent to the server on behalf of that client. Hence, the required data objects will follow the transaction to its new host site without requiring explicit requests.

  END;

END.

From the above pseudocode, it can be seen that the final decision about transaction-shipping is made by the clients. This ensures that the work done in load-sharing is distributed over all clients in the system. Off-loading the load-sharing effort from the server is important

because in situations of high load, the server may fall behind in serving object requests, maintaining load tables and forward lists.

4. Other enhancements for real-time transaction processing

This section describes the additional techniques that we have used to enhance the efficiency of the client-server architecture. These techniques are transaction decomposition [14, 33, 43], object migration planning, and object request scheduling. Transaction decomposition works above the load-sharing algorithm, and uses it to optimally execute decomposed sub-tasks. Object request scheduling and migration planning function underneath the load-sharing and attempt to satisfy clients' object requests in a deadline cognizant manner. This implementation hierarchy of all these functional components is shown in figure 2.

4.1. Transaction decomposition

Transactions in a database system operate on data in many different ways. In some cases, a transaction may not need to access all its required data objects simultaneously. One example query is shown in figure 3(a). If *employee* is an object database then this transaction can query each *employee* object independently and the complete answer can be constructed from individual query results. A more complex example is shown in figure 3(b). Although, the external SELECT cannot be decomposed, the nested component can be easily split into sub-tasks. *Manager* objects can be queried separately to determine which ones correspond to managers in the *Sales* department. The results of the nested query can then be combined and used to execute the external select.

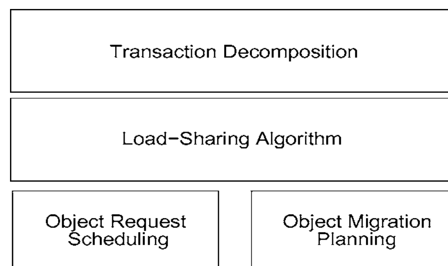

Figure 2. Implementation levels of the enhancement techniques.

```
SELECT employee.name
FROM employee
WHERE employee.salary > 10000;
```

(a) Simple Query

```
SELECT employee.name, employee.address
FROM employee
WHERE employee.id IN (SELECT manager.id
                      FROM manager
                      WHERE manager.department = "Sales");
```

(b) Nested Query

Figure 3. Two example queries.

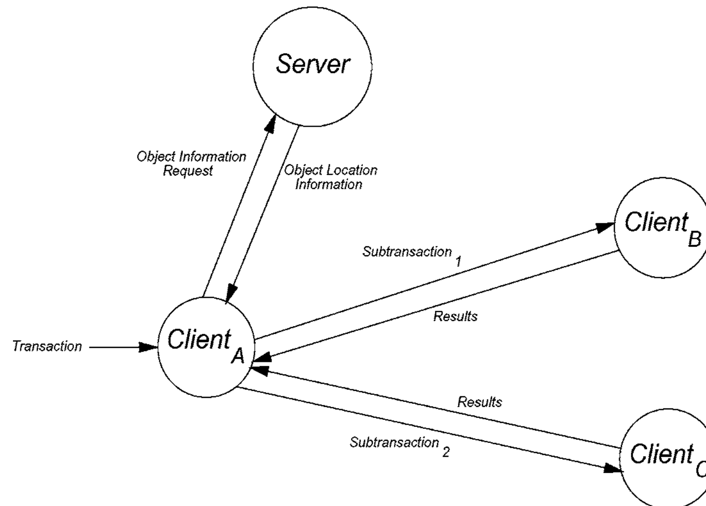

Figure 4. Transaction decomposition.

This idea of independent processing can be extended further to allow a transaction to execute at several sites that host its required data objects. Therefore, for the second query shown above, client sites that have cached *manager* objects could process the nested SELECT independently and then ship the results to the originating site to facilitate the execution of the outer SELECT. We term this concept of splitting transactions as *transaction decomposition*. This is conceptually different from *transaction chopping* as proposed in [44]. The primary objective in transaction chopping is to reduce locking delays in a transaction processing environment. This is done by breaking transactions up into smaller pieces that hold locks for smaller intervals of time. It is, of course, necessary to ensure that the execution of these smaller pieces guarantees the serializability of the original transactions. In contrast, transaction decomposition refers to the execution of the complete transaction on physically separate fragments of the requisite data and the synthesis of these partial results into the final answer.

In the CSD, transaction decomposition refers to the disassembly of multiple object requests from a client transaction and the quest to separately fulfill such isolated, and possibly independent, object requests (figure 4). Such decomposition can assist in the off-loading of a client site in two major ways: (i) when a client requests objects that have been cached in numerous sites (a situation reminiscent of data fragmentation in distributed databases [33]), and (ii) a set of pending client transactions can be considered all together and the client database manager can take advantage of the common object requests that these transactions may demonstrate [42, 43].

Transaction decomposition consists of three phases: request disassembly, materialization, and answer synthesis. For example, if a transaction needs objects already cached in four different locations then, the requesting client should communicate its data and processing needs to these four clients. This communication could be facilitated by the server which

is aware of the locations and the lock status of cached objects. Once the four independent clients process the necessary objects in parallel (materialization phase), they can forward the results to the requesting client (answer synthesis) for further processing and final transaction handling. In a slightly different scenario, some of the required objects can be retrieved from the server and the remainder from clients. In either scenario, transaction decomposition helps in off-loading the server's long-term memory at the expense of additional CPU processing and maintenance of meta-data.

Transaction decomposition can be incorporated into the load sharing algorithm in an elegant manner. If a transaction is decomposable then it can be executed as a set of independent sub-tasks. Each of these sub-tasks is treated individually as a separate transaction by the load sharing algorithm. For transaction decomposition to work in CSD environments, the server is required to keep track of the locations of all database objects in the system. A new meta-query mechanism may be necessary that allows clients to easily ask the server for a list of all clients that have the relevant data cached. Consider the query shown in figure 3(b), here the client at which the query was initiated could identify all the client sites that have *manager* objects cached and ship the inner SELECT to them. This information will allow the system to decide whether a transaction should be diverged to several sites, and which ones. However, it should be noted that general transaction decomposition is a difficult problem to solve. To help identify transactions that are candidates for decomposition, an *a priori* classification of transactions based on data access/update patterns may be established. Transactions decomposed in this manner have all the advantages that transaction-shipping provides along with the added benefit that each of the distributed processing tasks may take shorter time to process than the overall task at one site. Furthermore, each of these sub-tasks can be processed in parallel. The main disadvantage of such parallel subtask processing is that the failure of any subtask to meet the transaction deadline implies the failure of the entire transaction.

4.2. Object migration planning

The manner in which the movement of objects within a cluster is managed can play a significant role on the efficiency of the system. Such optimization is very easily possible when dealing with requests by multiple clients on the same object. The key idea here is to group the granting and release of locks in order to reduce the total number of messaging interactions required to satisfy client object requests [5].

In a conventional client-server database system, requests from n clients for a particular object will require one message from each of the clients to the server—n messages in all. The server will grant each request as soon as possible, and satisfying all n requests will require another n messages. At some point, each client will be required to release its lock on the object and return the object to the server. This phase will involve another n messages. Therefore, the total number of messages sent is $3n$. In a CSD that permits inter-transaction caching, a client may retain a cached object and lock until the server explicitly calls it back. The number of messages per object in this case will be 4 and the total number of message will be $4n$. Of course, clients may still release objects unilaterally if they want to (for instance, due to lack of space in their local caches). An example interaction is shown in

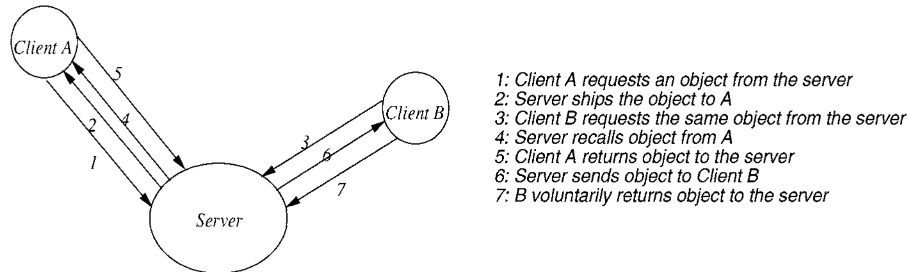

Figure 5. Message passing in the 2PL protocol.

figure 5 which indicates that satisfying one request each from $Client_A$ and $Client_B$ requires 7 messages.

Using the request grouping technique, we can reduce the total number of messages required to perform the above interaction. In this technique, the object server collects all the lock requests for each database object for a specified time interval (*collection window*) in an ordered list (*forward list*). At the end of the collection window, the lock is granted to the first transaction in the forward list and the object is shipped to the respective client along with the forward list. Appropriate information can also be placed in the forward list to indicate parallel read-only access to data. When this transaction commits, the client ships the object to the next client in the forward list. Finally, after the last transaction on the forward list completes, the object is returned to the server. In this scheme, the lock release of the previous client is combined with the lock grant of the next client. Therefore, for n requests on a database object within a collection window, 2PL will require $3n$ messages while the lock grouping protocol will require only $2n + 1$ messages. An interaction using lock grouping is shown in figure 6. Shipping the object to $Client_A$, then to $Client_B$ and back to the server requires 5 messages.

The forward list is prioritized according to an ordering rule. Possible orderings are First-Come First-Serve, transaction priority, and serving read requests first. In a real-time environment, the forward list can be prioritized based on the deadline of the requesting transactions. This information can be stored in the forward list and used to ignore transactions that have

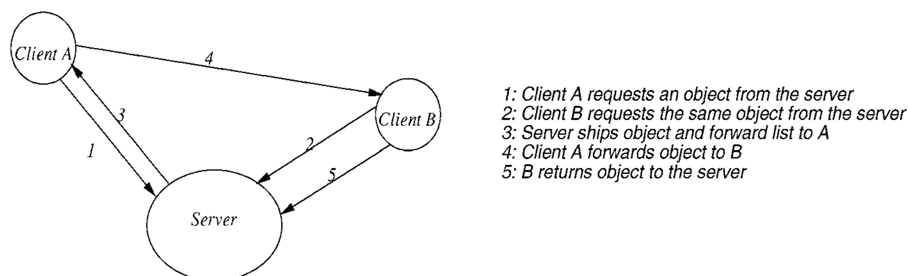

Figure 6. Message passing using the lock grouping protocol.

missed their deadlines. When accurate knowledge of transaction execution times is available, transactions that are expected to miss their deadlines can be ignored as well. However, using the forward list technique without modification can be inefficient in certain situations. For example, once an object has been shipped to a client along with its forward list, another transaction request may arrive at the server with a deadline that cannot now be satisfied. In order to try and reduce the number of transactions that miss their deadline due to this reason, we allow dynamic updates to the forward lists. For instance, consider a transaction T_1 that requests the object S_{91} from the server after the collection window is closed. S_{91} has already been shipped to the first client in its forward list and its present location is not known to the server precisely. To dynamically update the forward list for S_{91}, the server performs the following steps:

(i) the server identifies T_1's position in the forward list for S_{91} according to the current scheduling policy (i.e., ED or LS).
(ii) then it sends a message to the Client C_j that has the transaction just before T_1 (in the priority list) and instructs it to update the forward list. If this client is in a position to do so, then T_1's request will be served according to its priority.
(iii) in case C_j informs the server that it can no longer update S_{91}'s forward list, the server sends the forward list update to a small subset of the remaining clients in S_{91}'s forward list and the last one.

An example of a dynamic update to the forward list is shown in figure 7.

The object migration planning mechanism is independent of the load-sharing algorithm (figure 2); however, when a client requests an object's location from the server, the server refers to the object's forward list and reports the last client in that list, say C_j, as the location of the object. Using location information for all the objects requested by the transaction, the client decides whether to ship the transaction. If the client decides to process the transaction at a site other than C_j, then that client is added to the object's forward list. If the transaction is shipped to C_j then it can wait there for the object to arrive. It is our hypothesis that the system will have a lower number of missed deadlines using the dynamic update scheme for forward lists than without, and the number of messages passed will still be less than or equal to $3n$.

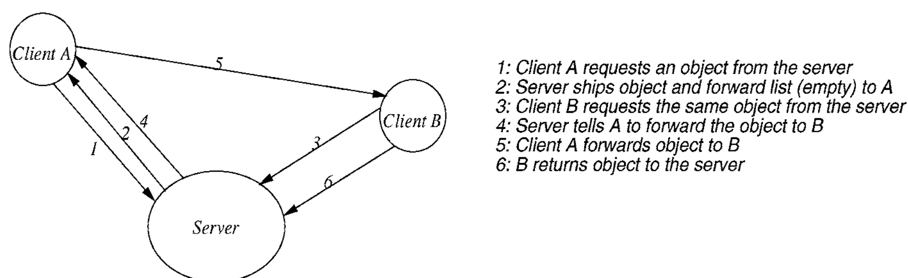

Figure 7. Dynamic forward list update.

4.3. Object request scheduling

The next technique that can be used in the context of real-time processing is the discipline with which object requests are satisfied by the server. In a non real-time environment, this discipline often is First-Come First-Serve. However, when client transactions bear deadlines, the queue of object requests can be prioritized according to the deadlines of the requesting transactions. Object requests by clients can convey the deadline information of the requesting transactions. Requests by transactions with earlier deadlines are satisfied before others. If a client transaction has already missed its deadline then the server can unilaterally decide not to ship the object to that client. In fact, if reasonably accurate estimates of transaction execution times are available, the object-server can decide whether to satisfy certain object requests at all, namely, requests by transactions that are expected to miss their deadlines. The object request scheduling mechanism functions independent of the load-sharing algorithm as it does not affect the operation of the LS-CS-RTDBS.

5. Evaluation objectives, testbed prototype creation and methodology

In this section, we evaluate the efficiency of the CS-RTDBS model for real-time transaction processing. This key feature of this evaluation is a performance comparison between the client-server and the centralized architecture for real-time transaction processing. We have developed prototypes of the CE-RTDBS, the basic data-shipping CS-RTDBS, and the CS-RTDBS that uses our load-sharing algorithm and other optimizations (LS-CS-RTDBS). In the following subsections, we describe the experimental setup and discuss the results of our prototype experiments. The main questions that our experimentation seeks to answer are:

- Is the performance of the CS-RTDBS comparable to that of the CE-RTDBS in a real-time processing environment?
- Do the load-sharing algorithm and the other techniques used in the LS-CS-RTDBS allow a better level of performance than the CS-RTDBS?
- How useful is the knowledge of precise transaction execution times in improving the efficiency of the three systems?

5.1. Experimental testbeds

The prototypes for all three systems have been written in C++ using the socket and thread libraries available on Sun Solaris 7. In all three packages, the server is a multi-threaded, connection-oriented program. At the beginning of each experiment the server listens on a socket for clients' connection requests. On receiving a connection request from a client, the server first establishes the required socket connections with that client, and then creates a new thread to manage that connection. This thread handles all future interaction with the client in question. Once a connection has been created between a client and the server, it is maintained for the entire duration of the experiment. This way the relatively high overhead of establishing socket connections between the clients and the server is incurred only once. We use TCP sockets for communication between the clients and their corresponding server

thread. This is because TCP provides a connection-oriented transport service, and handles packet loss and out-of-order delivery problems automatically.

In the LS-CS-RTDBS, clients are able to ship transactions and data objects to other clients. Since a client does not have open socket connections to all the other clients, there are two straightforward ways in which such transfers can be effected: (i) by opening a socket connection from the source client to the destination client, and (ii) by routing the shipped transactions and forwarded objects through the server. The first option is flexible but it incurs the overhead of establishing a socket connection for every object transfer. The second option avoids this overhead, but instead places an additional burden on the server. We avoid the disadvantages of both these methods by using a specialized directory server that maintains open socket connections to all clients. When a client wants to ship a transaction or an object to another client, it only needs to send it to the directory server which forwards it to its destination.

In our experiments, we assume that the database is a collection of objects, each of which has a unique object-ID. In order to create and manage such an object database, we have developed our own object-file manager (OFM). OFM provides the functionality needed to maintain a database consisting of equal-sized objects in a disk-resident file. It also incorporates a buffer manager that is used to control objects that have been read from the disk into memory. Whenever a request is made (by the client or server database) for an object, the buffer manager checks to see whether the requested object is already in the buffer pool. If so, the buffer manager simply returns a pointer to the object, otherwise the buffer manager reads in the requested object from disk into an empty slot in the buffer pool. If the buffer pool is full then the buffer manager frees a slot (if necessary, by writing the object it contains to disk) and then reads in the requested object into this slot.

Selecting the object to be evicted from the memory buffer is done using the *Simplified 2Q* algorithm [21]. *Simplified 2Q* improves over *Least Recently Used* (LRU) in that it tries to distinguish between hot and cold objects. It maintains two queues, one of which lists object accesses in the order in which they occurred. If an object is accessed once again, while its entry is still on the first queue, then it is identified as a likely hot page and moved to the second queue, which is managed in a LRU fashion. Therefore, a cold object that has very recently been brought into memory is still more likely to be evicted than a hot object (in the memory buffer) that has been accessed less recently. This is in contrast to pure LRU buffer replacement where a hot object may be removed from the memory buffer to make space for an incoming cold one. The *Simplified 2Q* buffer replacement algorithm is also used by clients to decide which objects should be returned to the server when they need to make space in their local caches.

A transaction in each of the prototypes is constructed by issuing calls to OFM functions to load and update database objects from the disk-resident database. Objects requested by transactions are read into memory using the *GetObject* method which retrieves the requested objects either from the disk or OFM memory buffer. Objects which are modified are marked as *dirty* using the *DirtyObject* function. Such objects are automatically written back to the disk file by the OFM buffer manager when the object is replaced in the buffer. The "processing" performed by each transaction is the calculation of products of random numbers until the prescribed transaction execution time has elapsed. When the transaction

completes, it releases the locks on the database objects. We assume that the server in the centralized system can process transactions twenty times faster than the clients in client-server systems.

The server in the CE-RTDBS and the clients in the CS-RTDBSs can execute multiple transaction concurrently. This is done by executing each transaction as a separate thread. Locking is used to ensure that all accesses to the database are serializable. The number of transactions that can be executed concurrently depends on the availability of memory buffer space and access to database objects. Transactions are scheduled and executed according to a real-time priority assignment algorithm. Since only the centralized server processes all transactions in the CE-RTDBS, it uses a single scheduler to prioritize all user-submitted transactions. In contrast, in the CS-RTDBS and LS-CS-RTDBS the clients perform transaction processing, and therefore, each client has its own local scheduler. Information about the load at the clients is communicated to the server piggybacked on object request and release messages. Since a client that has all its object working set cached may not be required to communicate with the server for long periods of time, we require that clients send their load information to the server at pre-determined intervals of no communication. The scheduling policies that we use in our experimentation are *Earliest Deadline First* (ED) and *Least Slack First* (LS). In order to use the LS scheduling policy we have assumed that the CPU processing time of each transaction is known. Of course, delays incurred due to locking and I/O overheads are not included in this estimate and are unpredictable. Figures 8 and 9 depict the CE-RTDBS and CS-RTDBS implementations respectively.

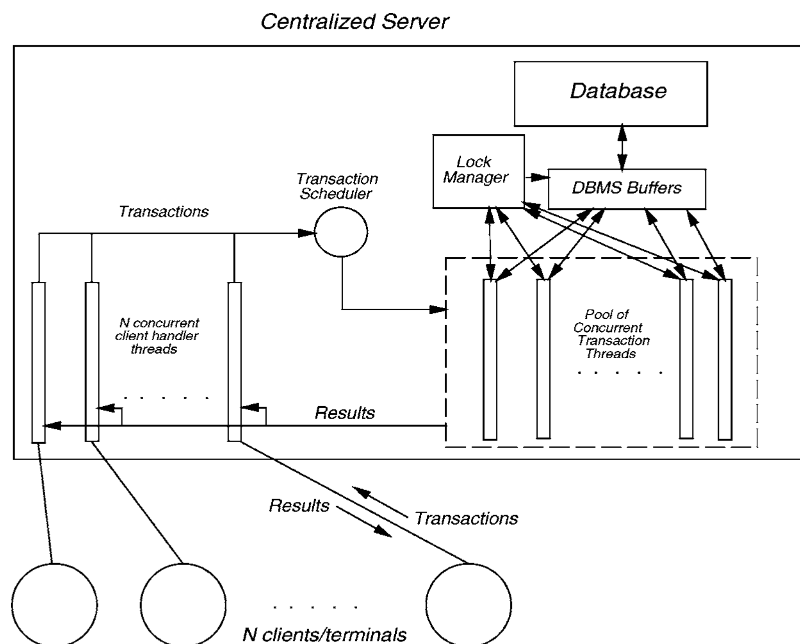

Figure 8. Implementation of the CE-RTDBS.

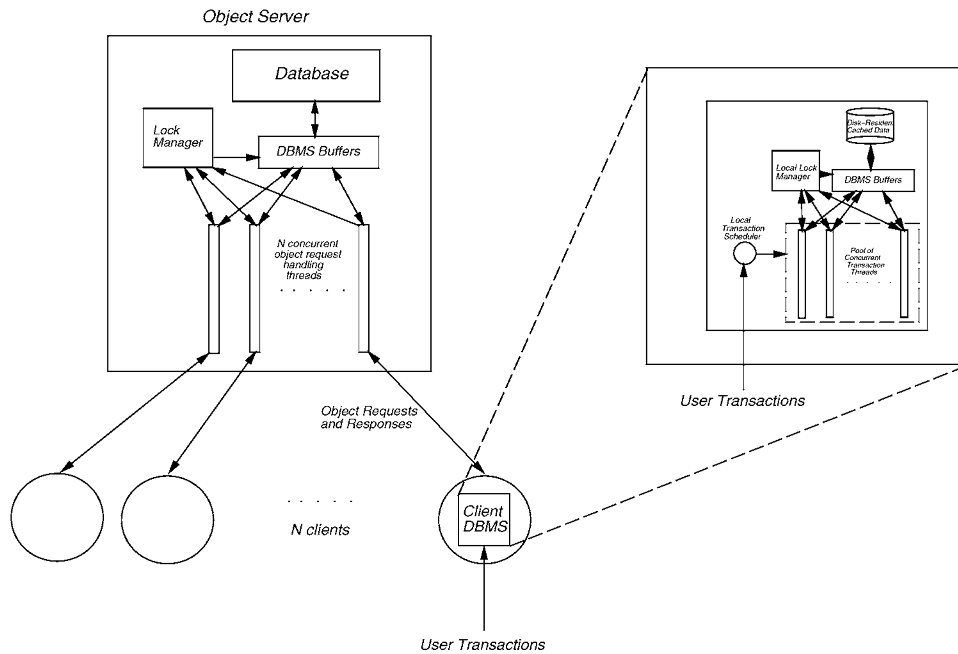

Figure 9. Implementation of the CS-RTDBS.

Lock managers are used in the centralized and client-server systems to ensure that concurrent transactions do not access objects in conflicting modes. The client-server RTDBSs require lock co-ordination at two levels. The global lock manager is located at the server and prevents clients from acquiring conflicting locks on data objects. Lock managers are also required at client sites to arbitrate between lock requests from local transactions. All lock managers use *wait-for* graphs to detect deadlocks [52]. 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 allows us to accommodate wait-for graphs within the lock tables resulting in considerable savings in memory space usage. When an object/lock request is received by the lock manager, this request is added to the request queue only if it does not cause a deadlock cycle. Accesses to the lock table and variables shared by multiple threads of the systems are controlled by means of Solaris mutual exclusion locking primitives.

5.2. Experimental methodology and parameters

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

Table 1. Parameters for the prototype experiments.

Parameter	Experimental Set 1	Experimental Set 2
Database size (objects)	50,000	50,000
Centralized RTDBS server main memory (objects)	25,000	25,000
CS-RTDBS server main memory (objects)	10,000	10,000
Client disk cache size (objects)	500	500
Client main memory size (objects)	500	500
Database access pattern	Localized-RW	Uniform
Average transaction inter-arrival time (seconds)	3	3
Average transaction length (seconds)	3	3
Average number of objects accessed by each transaction	10	10
Update selectivity	1%, 5%, 20%	1%, 5%, 20%

executed by itself on one workstation. The database clients were uniformly divided on the remaining workstations. The database used in our experiments contained a total of 50,000 objects. The size of each object was 4 Kb. The CE-RTDBS server was able to hold as much as 50% of the database in its main memory. The server in CS-RTDBS and LS-CS-RTDBS could hold up to 20% of the database in its main memory. Here, clients were able to host up to 10% of the server database locally: 5% in their disk cache and 5% in their main memory. The size of client caches is set so that benefits of database access locality can be exploited through the use of inter-transaction data caching. Table 1 lists the values of the parameters used in our prototype experiments. We believe that the faster processing ability and the much larger buffer space available in the CE-RTDBS give it an inherent advantage over the CS-RTDBSs.

We experimented with two different database access patterns. In one pattern, accesses to the database were distributed uniformly, i.e., a transaction could access any database object with equal probability. In the other access pattern—which we call *Localized-RW*—80% of each client's accesses were made to a particular portion of the database (hot region) according to the Uniform distribution. The cold region for each client is the remainder of the database including the frequently accessed areas of other clients; 20% of the client's accesses are made to this cold region and are distributed according to the Zipf distribution. The size of each client's hot area is equal to 1% of the database size. The design of the *Localized-RW* access pattern draws heavily from the HOTCOLD workload proposed in [12]. An important distinction, however, is our use of the Zipf distribution for calculating the access probabilities of objects in each client's cold region. We tested the three prototypes with 1, 5 and 20% update selectivities, i.e., the percentage of all object accesses that were updates.

In both systems, transaction arrivals at each client are determined by a Poisson arrival process with a mean inter-arrival time of 3 seconds. Ten percent of all submitted transactions

were assumed to be decomposable. The processing time for each transaction is generated according to an exponential distribution with an average of 3 seconds. (Table 1). For a transaction with a processing time of x, the deadline was randomly distributed in the range $[x, 3x]$. The wide range of possible transaction deadlines was used to reveal the performance difference between the ED and LS scheduling policies. The deadlines assigned to sub-transactions of a decomposed transaction were the same as that of the original transaction. In the future, we hope to develop a mechanism that assigns deadlines to sub-transactions depending on execution time estimates that are based either on the size of the data queried or on historical execution times. The average number of objects accessed by each transaction is 10 (exponentially distributed). According to this exponential distribution, a large number of transactions access 3–5 objects, but there are also some instances of transactions that access up to 100 objects. The next section describes the results of our experiments for each of the above three workloads. We ran our prototype systems five times for each test setting and every experiment was run for 3 hours. Results are collected and analyzed at the end of this period.

In the presentation of our experimental results, we consider the percentage of successful transactions, i.e., transactions that complete within their deadlines, as perhaps the most important measure of the performance of the models. This is because the success percentage is a critical factor that real-time systems should seek to optimize. We also present the overall number of messages passed and the average object response times in the CS-RTDBSs as additional metrics of performance. Finally, it should be noted that although we do not assign any importance to transactions that miss their deadlines, such transactions are not aborted. In our experiments, we reset their deadlines to a very large number (to represent infinity) and allow the transaction schedulers to decide their consequent order of execution—which is when there are no transactions with earlier deadlines contending for data and processing resources.

6. Testbed-derived experimental results

Experiment Set 1. In the first set of experiments, we evaluated the CE-RTDBS, CS-RTDBS, and the LS-CS-RTDBS with the *Localized-RW* database access pattern. In addition to using two different scheduling policies, we also varied the probability of updates. The percentage of transactions that completed within their deadlines in each of the three configurations for 1% update selectivity is shown in figure 10. The scheduling policy for each curve is shown in parentheses.

As the curves show, the LS-CS-RTDBS clearly outperforms the other two systems in this setting. This is true for both scheduling policies, ED and LS. We first compare the performances of the three systems for ED scheduling. For a small number of clients, the centralized system is able to complete a greater number of transactions successfully as compared to the CS-RTDBS. This is due to the faster processing ability of the centralized system and the low contention for database objects which allows the CE-RTDBS to exhibit a very high degree of concurrent transaction execution. However, as the number of clients increases, the performance of the CE-RTDBS system deteriorates rapidly. The primary reason for this deterioration is the fact that the CE-RTDBS uses only one scheduler to

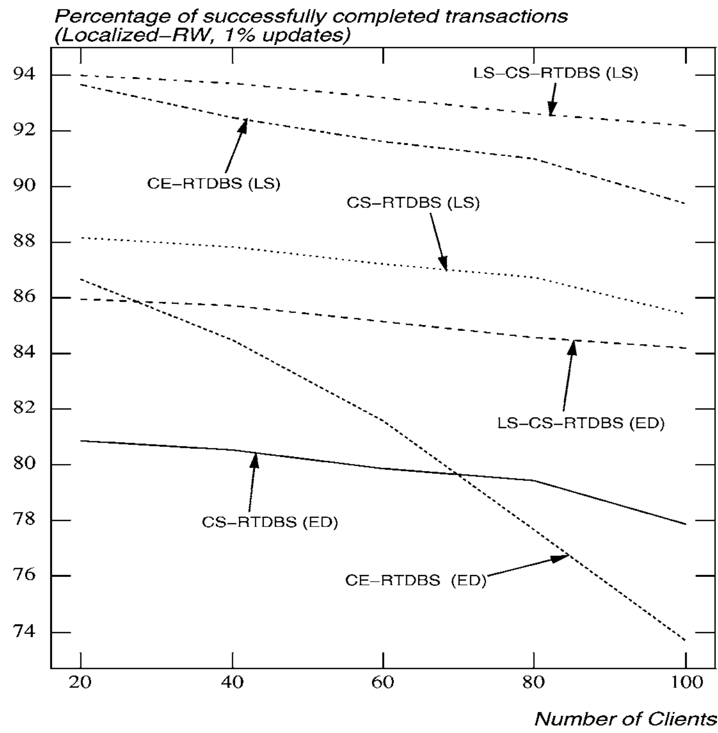

Figure 10. Percentage of transactions completed within their deadlines (localized-RW, 1% update selectivity).

assign priorities to transactions. Each bad scheduling decision has the potential to affect a very large number of other transactions. Once the number of clients is greater than 40, the centralized system does not perform as well as the CS-RTDBS. The locality in every client's data accesses and the low probability of updates (exclusive locks) means that a very high percentage of client data object requests can be satisfied without interacting with the server (Table 4). The low update selectivity also means that requests for objects from the server are satisfied in very short times (Table 3). This allows the CS-RTDBS to scale very well as the load on the system increases. Of the response times shown in Table 3, a very small fraction of the time is actually taken by the data transfer. Most of the time is spent waiting for clients/transactions to release their locks on the requested objects.

An important factor that causes transactions at a client (in the CS-RTDBS) to fail is the delay in obtaining objects/locks that have been acquired by other clients. The objective of our Load-Sharing CS-RTDBS (LS-CS-RTDBS) has been, from the very start, to reduce the number of transactions that fail due to this reason. The method in which this is done is to try to ship any transaction, that cannot immediately acquire its requisite objects/locks, to the site where the requisite data is resident. Doing this can only increase the probability of successful completion for that transaction. The load-sharing algorithm, along with the object migration policies, results in a significantly higher percentage of successful transactions in

Table 2. Messages passed in the client-server configurations for 100 clients (Localized-RW, 1% update selectivity).

Message type	CS-RTDBS	LS-CS-RTDBS
Object request messages (client to server)	268,217 (ED)	262,891 (ED)
	266,615 (LS)	259,628 (LS)
Objects sent (server to client)	265,186 (ED)	254,477 (ED)
	264,658 (LS)	251,371 (LS)
Object requests satisfied using forward lists (client to client)	–	12,115 (ED)
	–	10,913 (LS)
Objects recall messages (server to client)	131,053 (ED)	118,190 (ED)
	129,273 (LS)	115,628 (LS)
Objects returned (client to server)	129,574 (ED)	116,143 (ED)
	127,552 (LS)	114,005 (LS)

Table 3. Average object response times (seconds) in the CS-RTDBS and the LS-CS-RTDBS for the localized-RW access pattern and 1% update selectivity.

	CS-RTDBS		LS-CS-RTDBS	
Number of clients	Shared locks	Exclusive locks	Shared locks	Exclusive locks
20	0.034	0.061	0.033	0.053
60	0.047	0.139	0.043	0.090
100	0.079	0.261	0.072	0.208

the LS-CS-RTDBS than in the CS-RTDBS. The object migration techniques considerably reduced the number of messages passed in the system by scheduling and optimizing the movement of objects (Table 2). The number of requests that have been pushed along with the objects in forward lists is significant, and a corresponding reduction can be seen in the number of messages passed for object recalls and returns.

In order to assess the impact of each of the four optimization techniques individually, we also ran our experiments using only one technique at a time. In this setting, for 100

Table 4. Average cache hit rates in the CS-RTDBS and the LS-CS-RTDBS for the localized-RW access pattern with varying update selectivities.

	CS-RTDBS			LS-CS-RTDBS		
Number of clients	1%	5%	20%	1%	5%	20%
20	87.08	84.63	79.74	89.63	87.11	84.31
60	85.54	78.18	74.64	88.63	84.11	81.71
100	82.63	75.52	62.29	86.55	82.21	66.90

clients, Object Request Scheduling and the use of Object Migration Planning improved the efficiency of the CS-RTDBS by only about 1% each. The use of Transaction Decomposition provided a 3% increase in the percentage of successful transactions. The load sharing algorithm was most beneficial and resulted in a 4.5% increase in the percentage of successful transactions. It should be noted that the sum of the improvements offered by these individual techniques is greater than the 6% higher efficiency that LS-CS-RTDBS (ED) has over the simple CS-RTDBS (ED) (figure 10). This is because the sets of transactions helped by each technique are not independent of each other.

In order to use the LS scheduling strategy we assumed that the CPU execution time of each transaction is known precisely. As described in Section 3, the knowledge of transaction execution times allows for qualitative improvements in transaction scheduling, object request prioritization, and object migration planning. Therefore, as figure 10 shows, the efficiency of all three systems is improved to a great extent. The availability of transaction execution times allows transaction schedulers in each system to avoid tasks that are expected to miss their deadlines. The improved scheduling allows the CE-RTDBS to perform better than the CS-RTDBS even for large numbers of clients. Resources wasted in processing infeasible transactions can be avoided, and therefore, the greater processing capability of the CE-RTDBS proves to be more effective than the distributed scheduling in the CS-RTDBS. The LS-CS-RTDBS is still the most efficient architecture. Knowing the exact transaction processing times improves the decisions made by the load-sharing algorithm. The effectiveness of the object migration technique is also increased accordingly. Forward lists can now be managed in a much better way; dynamic modifications to the forward lists are easier as the server is now able to estimate the current location of a forwarded object with much more precision. The effects of these improvements can be seen in the reduction in the number of messages and objects shipped in the LS-CS-RTDBS (Table 2).

Once again, when used individually, only transaction decomposition and the load-sharing algorithm have a significant impact on the efficiency of the LS-CS-RTDBS, i.e., a 2.5 and 4% improvement respectively in the percentage of successful transactions. Object Request Scheduling and Object Migration Planning contribute approximately 1% improvements in the efficiency only. Overall, the LS-CS-RTDBS (LS) demonstrates slightly more than a 6% improvement in efficiency over the CS-RTDBS (LS).

The performance of the three models for an update selectivity of 5% is shown in figure 11. As the curves show, the increased contention resulting from a higher probability of updates affects the performance of all three systems adversely. This is true irrespective of the scheduling policy used. In the centralized system, the effect of this increased contention is a reduction in the degree of transaction concurrency. The CS-RTDBS is affected the most by the increased locking conflicts. In cases of lock conflicts between clients, the data object has to be returned to the server and re-fetched when needed again. This results in longer blocking periods for client transactions. Particularly, when an object in a client's frequently accessed region has to be returned to the server, the delay in re-fetching it can affect several transactions.

The LS-CS-RTDBS can avoid such delays in a large number of cases. Rather than wait for the data be sent to the client where the transaction was initiated, the transaction itself is shipped to the client which has presently locked the data. For either scheduling discipline,

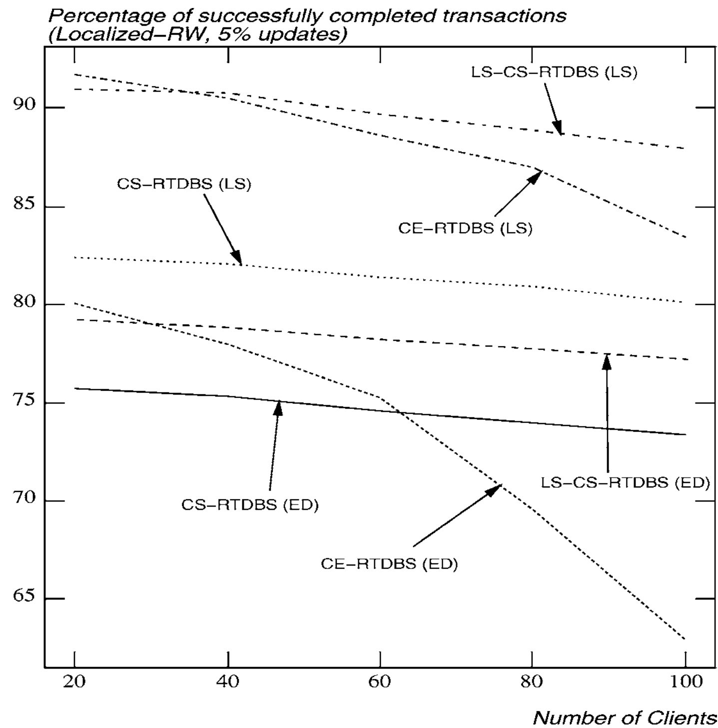

Figure 11. Percentage of transactions completed within their deadlines (localized-RW, 5% update selectivity).

the flexibility in the execution of transactions allows the LS-CS-RTDBS to outperform the CS-RTDBS and centralized system (once the number of clients is greater than 20). In case of ED scheduling, for a large number of clients, the CS-RTDBS is more efficient that the centralized system. On the other hand, when LS is used to prioritize transactions, the CE-RTDBS is clearly better. As the load increases, the two client-server systems are able to offer a much more stable level of performance than the CE-RTDBS.

Figure 12 shows the performance of the three models when the update selectivity is 20%. The overall performance trends match those seen in the earlier scenario. For either scheduling strategy, the CS-RTDBS and the LS-CS-RTDBS demonstrate a very small deterioration in performance as the load increases. This is also true of the CE-RTDBS for LS scheduling. However, for ED scheduling, the CE-RTDBS's efficiency degrades very rapidly. Here, even though the CE-RTDBS does better than the LS-CS-RTDBS initially, the rapid fall in its performance makes it worse when the number of clients becomes larger than 40. The very high update selectivity has a significant adverse effect on the CS-RTDBS. A very large percentage of its object requests (almost all of those outside its frequently accessed range) cannot be granted by the server immediately. Similarly, an increased number of objects have to be given up and re-fetched from the server. These delays are instrumental in causing the CS-RTDBS to perform much worse than the LS-CS-RTDBS, and for less than 80 clients, worse than the CE-RTDBS as well. It is only the locality in each client's data accesses

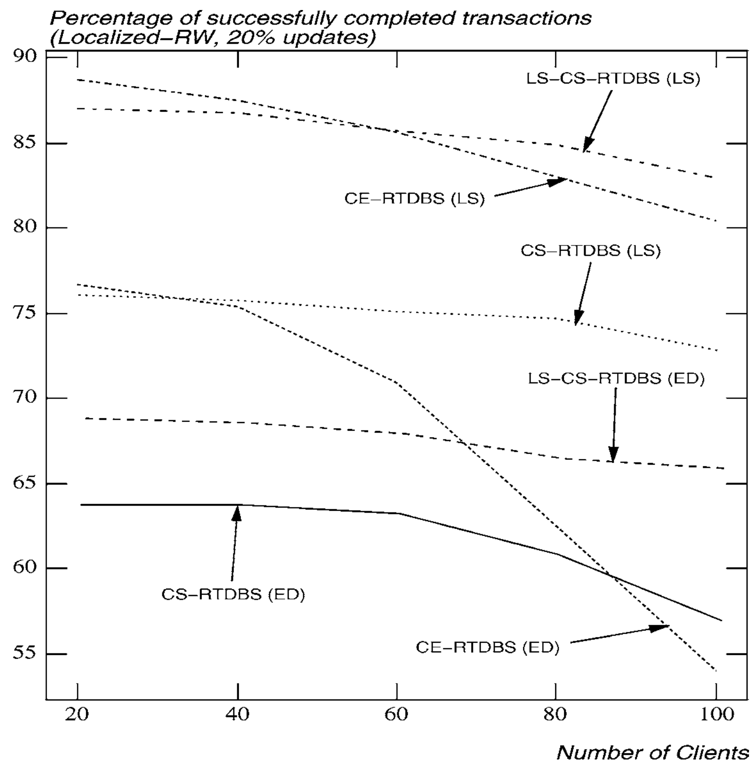

Figure 12. Percentage of transactions completed within their deadlines (Localized-RW, 20% update selectivity).

that causes the CS-RTDBS to do better than the CE-RTDBS when the number of clients is 100.

From the graph, we can see that the performance of the LS-CS-RTDBS is comparable to that of the CE-RTDBS even in the presence of a very high percentage of modifications. Once the number of clients is greater than 40, the LS-CS-RTDBS demonstrates a higher level of efficiency that the CE-RTDBS. The use of the load-sharing algorithm and the object migration planning techniques can be seen to have a considerable impact on the performance of the client-server model. For LS scheduling, the LS-CS-RTDBS completes almost 10% more transactions successfully than the basic data-shipping CS-RTDBS. This is a significant improvement, especially in a real-time environment.

Now, given the higher percentage of updates, the Object Request Scheduling and Object Migration Planning techniques are able to provide a noticeable improvement in the performance of the LS-CS-RTDBS, i.e., 2.5 and 3.5% respectively. This is because the higher percentage of updates creates many more cases where the object-request scheduler can resolve a lock conflict in favor of the transaction that is more likely to meet its deadline. The same is true in the case of object migration planning. More numerous exclusive locks mean that planning the movement of data objects within the cluster acquires greater importance. Load sharing is still the single most beneficial strategy and its use improves the efficiency

of the LS-CS-RTDBS by almost 8%. Transaction decomposition comes in second with an improvement in efficiency of 5%.

Experiment Set 2. In the second set of experiments, we evaluated the three real-time database configurations with uniformly distributed database accesses. Figure 13 shows the performance of the three systems when the update selectivity is 1%. As the plots show, the efficiency of the centralized system is not affected very much by the change in the database access distribution. This is because the CE-RTDBS does not incur the overhead of data transfers between client sites. In fact, the uniformly spread database accesses reduce the data contention and lock conflicts experienced by the CE-RTDBS. This is not the case with the client-server systems. Uniformly spread data accesses reduce the effectiveness of the client caches dramatically. The reduced transaction access locality at each site means that transactions are required to block much more frequently as they wait for their required data to be fetched from the server. The average times for object requests to be satisfied are shown in Table 5. These object response times are noticeably higher than those of the *Localized-RW* access distribution. Cache replacement rates in the client-server systems are also high, thus, resulting in an increased amount of network data transfer. Even so, for ED scheduling, the LS-CS-RTDBS is able to perform better than the CE-RTDBS

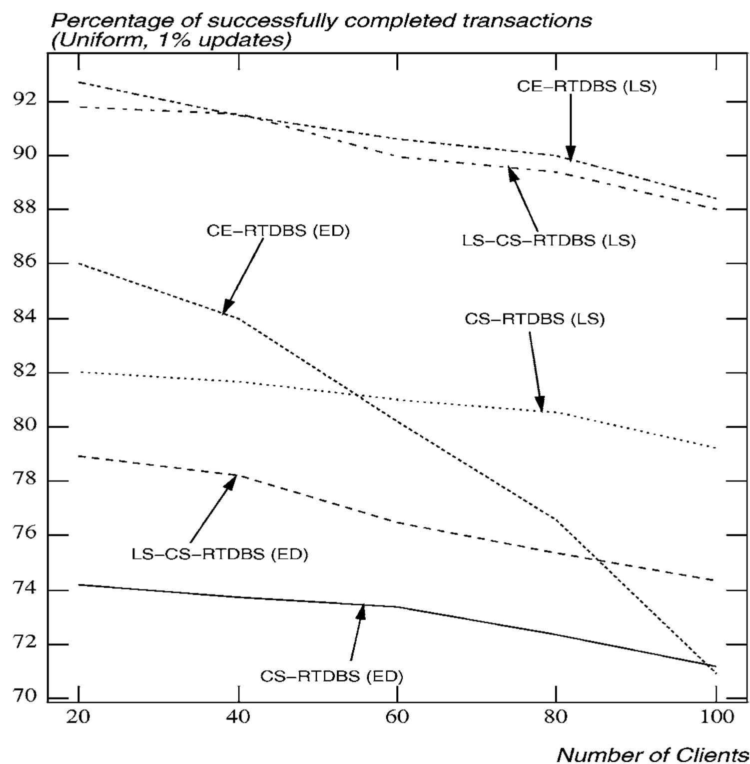

Figure 13. Percentage of transactions completed within their deadlines (uniform, 1% update selectivity).

Table 5. Average object response times (seconds). in the CS-RTDBS and the LS-CS-RTDBS for the uniform database access pattern and 1% update selectivity

	CS-RTDBS		LS-CS-RTDBS	
Number of clients	Shared locks	Exclusive locks	Shared locks	Exclusive locks
20	0.044	0.089	0.044	0.073
60	0.081	0.222	0.072	0.192
100	0.171	0.549	0.128	0.431

when the number of clients becomes greater than 80. With its load-sharing and object migration abilities, the LS-CS-RTDBS is also easily able to outperform the CS-RTDBS, and is also more efficient than the centralized system when the number of clients is greater than 60.

For LS scheduling, the CE-RTDBS demonstrates the best performance of the three system models. As mentioned earlier, knowing transaction execution times precisely allows the CE-RTDBS to avoid the scheduling inefficiencies that can result from having a single centralized scheduler. Transactions that are expected to miss their deadlines can be terminated without any time wasted in unnecessary processing. Although, the CS-RTDBS can also benefit from this, the lack of locality in each client's data accesses results in increased transaction blocking. As a consequence, the efficiency of the CS-RTDBS is never able to match that of the CE-RTDBS. The LS-CS-RTDBS can alleviate the problems caused by the uniform access distribution to some extent. Transactions that do not have their requisite data available locally can migrate to other sites. Due to this flexibility, the LS-CS-RTDBS can provide a level of performance comparable to the CE-RTDBS.

For an update selectivity is 5%, the CE-RTDBS is clearly the better alternative for real-time transaction processing. The only exception is when the transaction execution times are not known (ED scheduling) and the number of clients is high. The results for this experimental setting are shown in figure 14. As seen in the earlier plots, the efficiency of the CE-RTDBS degrades rapidly as the number of clients is increased. This is, however, still better than the efficiency of the two client-server systems. The higher percentage of exclusive locks requested by the transactions causes increased transaction blocking due to lock conflicts. As there is no locality in the clients' data accesses, most of the transaction data requests cannot be satisfied at the client and have to be forwarded to the server. In the CS-RTDBS, for 20 clients (ED scheduling), the average cache hit percentage at the clients is a mere 11%, and when the number of clients is one hundred, the cache hit rate is even lower at 7.4%.

Increasing the update selectivity to 20% has a very harsh effect on the CS-RTDBS and the LS-CS-RTDBS. The performance of these systems, in the face of virtually no database access locality and a very high probability of updates, is significantly lower than that of the CE-RTDBS. In the CS-RTDBS, the delays in obtaining data objects from the server become intolerably high which causes a significantly considerable proportion of transactions to miss their deadlines. The LS-CS-RTDBS is able to avoid many such missed transaction deadlines,

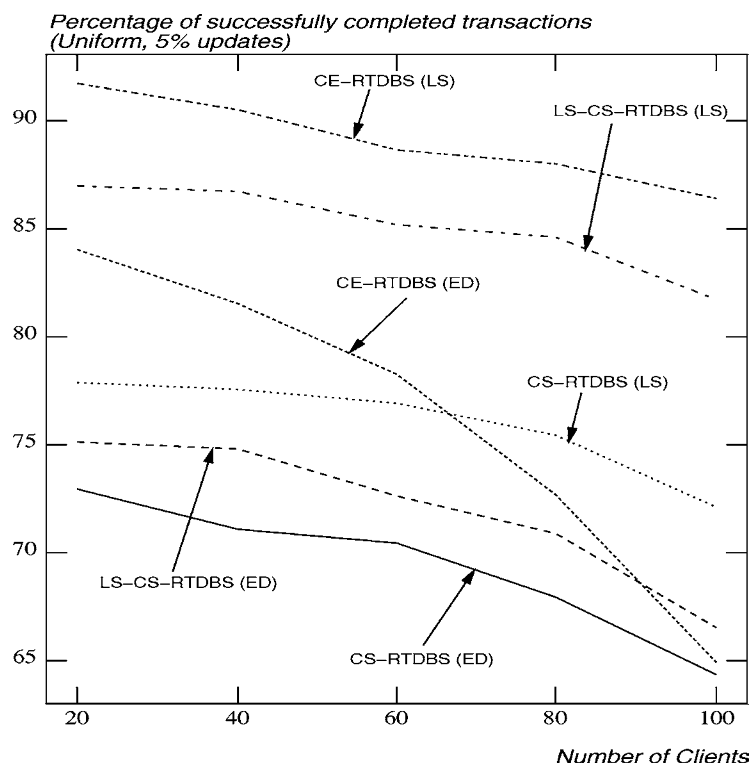

Figure 14. Percentage of transactions completed within their deadlines (uniform, 5% update selectivity).

but the uniformly scattered data means that very few transactions can be shipped to sites which have a greater number of data objects available locally. In contrast, the CE-RTDBS is affected only by the increased contention for data. Therefore, the performance of the CE-RTDBS is not significantly lower than for the Localized-RW access pattern. The results for this set of experiments are shown in figure 15.

7. Related work

In this section, we describe related work in the areas of load-balancing and real-time scheduling and processing. Global load balancing in multi-computer clusters requires up-to-date knowledge about the load on the participating sites. Collecting and cataloging this information may constitute some overhead on system resources. Many schemes have been suggested for load sharing in shared memory multi-processor machines [6, 15, 29, 38, 40, 51]. Since the data is visible to all processors, the load sharing exclusively concerns CPU loads. The majority of these approaches attempt to determine the global average load, and locate the least and most utilized nodes in the system using a minimum of communication steps. In [6, 15], the strategy for each node is to query only a small number of neighbors at periodic

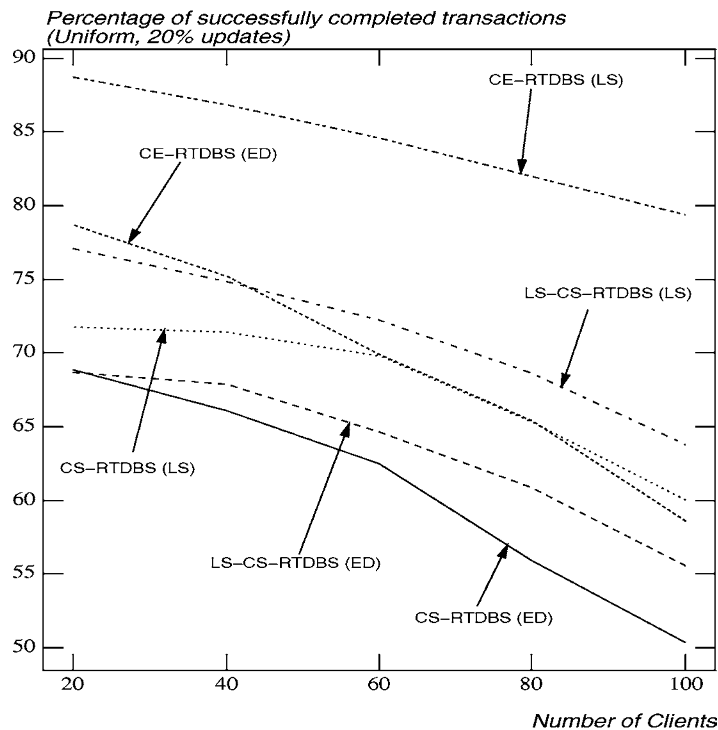

Figure 15. Percentage of transactions completed within their deadlines (uniform, 20% update selectivity).

intervals with the goal of keeping its own information up-to-date. An alternative scheme is to equalize the load on pairs of sites at a time [40]. A fully distributed load balancing algorithm for a connected N-processor network is proposed in [51]. Here, task migration is used to try and equalize the load on the processors in the system. A new communication protocol is also presented that guarantees that task migration will occur whenever there is at least one lightly loaded processor and at least one heavily loaded processor in the processor cluster.

Several algorithms for adaptive load sharing in multi-computer systems have been suggested [10, 45]. In [10], the notion of being adaptive is related to the frequency with which the load sharing task is run. In such methods, a site i executes the load sharing task with a probability $1/L$, where L is the number of pending transactions. If L is large, then the site is heavily loaded; otherwise, the site is lightly loaded and assignments are frequently solicited from neighboring nodes. In [45], Shin and Chang consider receiver-initiated load sharing in a multi-node real-time system. Unlike earlier load sharing techniques, each node maintains state information for only a small number of neighboring nodes (*buddy-list*) and overloaded nodes transfer tasks to under-loaded nodes in accordance with a *preferred list*. The preferred list for each node is an ordering of nodes that tasks are to be transferred to when the node is overloaded. Here, task queue-length thresholds are used to classify nodes in the system as either under-loaded, medium-loaded or overloaded. Each node broadcasts its load state

to the nodes in its buddy list only when the state changes (according to the queue-length thresholds); this eliminates the need for periodic polling between nodes. Conceptually, our approach to load sharing closely resembles this, but an additional factor that we consider is the presence of the appropriate data at the candidate load sharing nodes. The work in [45] is further extended in [46, 47]. Bayesian decision theory is used to enhance possibly incomplete and out-of-date load state information about nodes. Consequently, the possibility of failure of real-time tasks due to inaccurate information about the load at receiver nodes is significantly reduced.

Real-time systems have traditionally exploited available information about the submitted tasks so as to guarantee their time-constrained completion. This information includes the task's deadline, its expected processing time and I/O requirements, and the priority assigned to it. Scheduling techniques have been proposed which make use of this information in order to maximize the number of transactions which complete within their deadlines [1, 30, 35]. Optimization of assignment of tasks in distributed soft real-time environments has been studied in [26]. One of the first detailed experimental evaluations of real-time transaction processing is presented in [20]. In this work, Huang et al. have identified the CPU scheduling algorithm as having the most impact on the efficiency of the RTDBS. They also make the observation that overheads in locking and message communication are non-negligible and cannot be ignored. The latter conclusions are particularly applicable to a CS-RTDBS where the transaction processing is performed by a cluster of shared-nothing nodes. Kalogeraki et al. [23] propose the Cooling and Hot-Spot algorithms to attain load balancing in a network of processors for mostly CPU-intensive tasks. Effective profiling of run-time application object behavior helps determine resource overloading and inability for tasks to meet their deadlines prompting object migration for load sharing.

Techniques for load sharing have also been proposed in distributed database systems. Application of an economic model to adaptive query processing and dynamic load-balancing in a non real-time context was proposed in [50]. Load-balancing heuristics for use in a distributed real-time system are described in [49]. These heuristics attempt to locate candidate nodes that can complete tasks whose deadline cannot be met by at the node that they were initiated. In [8], Bestavros and Spartiotis describe a heuristic that migrates tasks from one node of a distributed real-time system to another if the latter node offers a higher probability of successful completion for the task. Load-profiling as a means of locating nodes with a sufficient surplus of computation cycles to successfully complete non-periodic tasks is examined in [7]. The problem of determining the order in which periodic update transactions should be considered for period and deadline assignments is discussed in [56]. A novel approach called *More-Less* is proposed in which updates occur with a period that is more than that proposed by earlier approaches [36] but with a relatively shorter deadline. The objective is to minimize the workload induced by update transactions.

The primary distinction between the works described above and our load-sharing algorithm is that we examine the problem of scheduling real-time transactions in a database environment. In this setting, the availability of database objects/locks is essential for a transaction to proceed. The load-sharing algorithm we have proposed uses information about the location of data and available client resources in order to minimize the percentage

of missed deadlines. In contrast to conventional approaches, we do this by moving the transactions or the data or both to the site that is most likely to successfully complete the transaction.

8. Conclusions

The use of client-server systems for database computing is pervasive. Powerful client machines coupled with available high speed networks have made efficient real-time processing in such systems a viable option. In this paper, we describe an algorithm for distributing the transaction processing load among the client sites in a CS-RTDBS. This load-sharing algorithm transfers processing tasks to clients that can provide immediate access to the required data and are less loaded. In a client-server setting, it is possible to maintain up-to-date information about the availability of data and spare processing capacity at the clients without incurring a significant overhead. In addition to the load-sharing algorithm, we have proposed a set of techniques that enhance the real-time processing capabilities of a client-server database. These techniques are: transaction decomposition, object migration planning, and object request scheduling.

In order to evaluate our load sharing algorithm, we have developed detailed prototypes of the centralized (CE-RTDBS), basic data-shipping client-server (CS-RTDBS), and load-sharing client-server (LS-CS-RTDBS) real-time databases. We state our conclusions separately for each of the two assumptions: (i) the CPU processing time of transactions was not known at all, and (ii) the CPU processing time of each transaction was known precisely. From the results that we obtained when operating under the first assumption, we conclude that:

- The client-server systems provide a very consistent level of performance as compared to the centralized real-time database system. As the number of clients increase, the performance of the CE-RTDBS degrades very rapidly while those of the CS-RTDBS and LS-CS-RTDBS show very little deterioration. We believe that the primary reason for the very rapid performance degradation in the CE-RTDBS is the fact that it uses a single scheduler to assign priorities to all transactions. Every badly scheduled task has the potential to affect and delay many subsequent tasks.
- The use of load-sharing and object migration policies in the LS-CS-RTDBS significantly improve its efficiency over that of the CS-RTDBS. In fact, the LS-CS-RTDBS completes 10% more transactions than the basic CS-RTDBS under the Localized-RW database access distribution with a 20% update selectivity. This is a considerable improvement in a real-time environment.
- An increase in the probability of updates affects the client-server systems more than the centralized one. However, the load-sharing technique enables the LS-CS-RTBDS to avoid unnecessary data movement and offer a much better level of performance than the basic data-shipping CS-RTDBS.
- When the accesses to the database demonstrate a degree of locality the performance improvements achieved by adapting a client-server system to perform load sharing outweigh the incurred overheads. In fact, the LS-CS-RTDBS can outperform its centralized

counterpart even in the presence of a very small number of clients. As expected, the absence of any data access locality causes the efficiency of the client-server DBSs to be significantly reduced.

When the second assumption was held to be true, we were able to use more prescient techniques for scheduling transactions and satisfying client object requests. The conclusions that we draw from this experimentation are:

- The CE-RTDBS is now a significantly better operating environment than the data-shipping CS-RTDBS. This is true for both workloads and all numbers of clients. This is because the CE-RTDBS system can make use of available transaction execution times to eliminate the disadvantage of having a single centralized scheduler. Transactions that are tardy or infeasible need not be scheduled at all.
- For the Localized-RW access pattern, LS-CS-RTDBS is still a viable option to the CE-RTDBS. Even for the Uniform data access pattern, the efficiency of LS-CS-RTDBS is comparable to that of the centralized system for a low update selectivity (1%). Once the update selectivity is high, the CE-RTDBS proves to be the best option for real-time transaction processing.

This paper investigates the real-time transaction processing efficiency of CSDs that use pessimistic locking to perform concurrency control. In the future, we plan to extend the study by using optimistic as well as speculative transaction processing techniques. Another important area for future research is the possibility of having transactions with different levels of importance. This is a very practical aspect of real-time transaction processing that we believe needs to be studied in detail.

Acknowledgments

We would like to thank the reviewers for their comments and suggestions that helped us improve the presentation of our work.

References

1. R. Abbott and H. Garcia-Molina, “Scheduling real-time transactions: A performance evaluation,” ACM Transactions on Database Systems, vol. 17, no. 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. Akamai.com, “Internet infrastructure service: An essential component for business success on the internet,” http://www.akamai.com/en/html/services/white_paper_library.html, 2001.
4. 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.
5. S. Banerjee and P. Chrysanthis, “Data sharing and recovery in gigabit-networked databases,” in Proceedings of the Fourth International Conference on Computer Communications and Networks, Las Vegas, NV, September 1995.

6. K. Baumartner, R. Kling, and B. Wah, "Implementation of GAMMON: An efficient load balancing strategy for a local computer system," in Proceedings of the International Conference on Parallel Processing, Aug. 1989, vol. 2, pp. 77–80.
7. A. Bestavros, "Load profiling: A methodology for scheduling real-time tasks in a distributed system," in Proceedings of the IEEE International Conference on Distributed Computing Systems, Baltimore, MD, USA, May 1997.
8. A. Bestavros and D. Spartiotis, "Probabilistic job scheduling for distributed real-time applications," in Proceedings of the First IEEE Workshop on Real-Time Applications, New York, NY, USA, May 1993.
9. 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.
10. F. Bonomi and A. Kumar, "Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler," IEEE Transactions on Computers, vol. 39, no 10, pp. 1232–1250, 1990.
11. A. Bouguettaya, B. Benatallah, L. Hendra, M. Ouzzani, and J. Beard, "A dynamic architecture for organizing and querying web-accessible databases," IEEE Transaction on Knowledge and Data Engineering, vol. 12, no. 5, pp. 779–801, Sept./Oct. 2000.
12. 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.
13. R. Cattell, Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison Wesley: Reading, MA, 1991.
14. D. Chatziantoniou, "Ad hoc OLAP: Expression and evaluation," in Proceedings of the 15th International Conference on Data Engineering, IEEE Computer Society: Sydney, Austrialia, March 1999.
15. Y. Chow and W. Kohler, "Models for dynamic load balancing in a heterogenous multiple processor system," IEEE Transactions on Computers, vol. 28, no. 5, pp. 334–361, 1979.
16. A. Delis and N. Roussopoulos, "Performance comparison of three modern DBMS architectures," IEEE Transactions on Software Engineering, vol. 19, no. 2, pp. 120–138, 1993.
17. 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, 1990, pp. 107–121.
18. D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," IEEE Transactions on Software Engineering, vol. 12, no. 5, pp. 662–675, 1986.
19. A. Gal, S. Kerr, and J. Mylopoulos, "Information services on the web: Building and maintaining domain models," International Journal of Cooperative Information Systems, vol. 8, no. 4, pp. 227–254, 1999.
20. J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," in Proceedings of the 10th Real-Time Systems Symposium, Santa Monica, CA, December 1989.
21. T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, Sept. 1994, pp. 439–450,
22. V. Kalogeraki, A. Delis, and D. Gunopulos, "Peer-to-peer architectures for scalable, efficient and reliable media services," in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.
23. V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser, "Dynamic migration algorithms for distributed object systems," in 21st IEEE International Conference on Distributed Computing Systems, Phoenix, AZ, April 2001.
24. 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, Sept. 1997.
25. V. Kanitkar and A. Delis, "Site selection for real-time client request handling," in Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Austin, TX, USA, May/June 1999.
26. 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.
27. A. Keller and J. Basu, "A predicate-based caching scheme for client-server database architectures," The VLDB Journal, vol 5, no. 1, pp. 35–47, 1996.

28. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The objectStore database system," Communications of the ACM, vol. 34, no. 10, 1991.
29. S. Lee, C. Yang, S. Tseng, and C. Tsai, "A cost-effective scheduling with load balancing for multiprocessor systems," in Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, Beijing, China, May 2000.
30. V. Lortz, K. Shin, and J. Kim, "MDARTS: A multiprocessor database architecture for hard real-time systems," IEEE Transactions on Knowledge and Data Engineering, vol. 12, no. 4, pp. 621–644, 2000.
31. S. Milliner, A. Bouguettaya, and M.P. Papazoglou, "A scalable architecture for autonomous feterogeneous Database Interactions," In Proceedings of 21th International Conference on Very Large Data Bases, Morgan Kaufmann: Zurich, Switzerland, September 1995.
32. M. Mutka and M. Livny, "The available capacity of a privately owned workstation Environment," Performance Evaluation, vol. 12, no. 4, pp. 269–284, 1991.
33. S. Nural, P. Koksal, F. Ozcan, and A. Dogac, "Query decomposition and processing in multidatabase systems," in Proceedings of the OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis, July 1996.
34. 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, New Orleans, LA, USA, Feb.-March 1996, pp. 344–351.
35. K. Ramamritham, "Allocation and scheduling of complex periodic tasks," in Proceedings of the 10th IEEE International Conference on Distributed Computing Systems, Paris, France, 1990.
36. K. Ramamritham, "Real-time databases," Distributed and Parallel Databases, vol. 1, no. 2, pp. 199–226, 1993.
37. K. Ramamritham, "Where do deadlines come from and where do they Go?" International Journal of Database Management, vol. 7, no. 2, pp. 4–10, Spring 1996.
38. K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 2, 1990.
39. N. Roussopoulos and H. Kang, "Principles and techniques in the design of ADMS±," IEEE Computer, vol. 19, no. 2, pp. 19–25, 1986.
40. L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "A simple load balancing scheme for task allocation in parallel machines," in ACM Symposium on Parallel Algorithms and Architectures, July 1991.
41. 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.
42. T. Sellis, "Intelligent caching and indexing techniques for relational database systems," Information Systems, vol. 13, no. 2, pp. 175–185, 1988.
43. T. Sellis, "Multiple–query optimization," ACM Transactions on Database Systems, vol. 13, no. 1, 1988.
44. D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction chopping: Algorithms and performance studies," ACM Transactions on Database Systems, vol. 20, no. 3, pp. 325–363, 1995.
45. K. Shin and Y. Chang, "Load sharing in distributed real-time systems with state-change broadcasts," IEEE Transactions on Computers, vol. 38, no. 8, pp. 1124–1142, 1989.
46. K. Shin and C. Hou, "Analytical models of adaptive load sharing schemes in distributed real-time systems," IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 7, pp. 740–761, 1993.
47. K. Shin and C. Hou, "Design and evaluation of effective load sharing in distributed real-time systems," IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 7, pp. 704–719, 1994.
48. V. Shkapenyuk and T. Suel, "Design and implementation of a high-performance distributed web crawler," in IEEE International Conference on Data Engineering, San Jose, CA, 2002.
49. J. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," IEEE Transactions on Computers, vol. 34, no. 12, pp. 1130–1143, 1985.
50. M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," The VLDB Journal, vol. 5, no. 1, 1996.
51. T. Suen and J. Wong, "Efficient task migration algorithm for distributed systems," IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 4, pp. 488–499, 1992.
52. A. Thomasian, "Concurrency control: Methods, performance, and analysis," ACM Computing Surveys, vol. 30, no. 1, pp. 70–119, 1998.

53. O. Torbjornsen and S. Bratsberg, “Designing an ultra highly available DBMS,” in Proceedings of ACM-SIGMOD Conference, Dallas, TX, May 2000.
54. Y. Wang and L. Rowe, “Cache consistency and concurrency control in a client/cerver DBMS architecture,” in Proceedings of the 1991 ACM SIGMOD Conference, Denver, CO, May 1991.
55. K. Wilkinson and M. Neimat, “Maintaining consistency of client-cached data,” in Proceedings of the 16th International Conference on Very Large Data Bases, Brisbane, Australia, August 1990, pp. 122–133.
56. M. Xiong and K. Ramamritham, “Deriving deadlines and periods for real-time transactions,” in Proceedings of the 20th Real-Time Systems Symposium, Phoenix, AZ, Dec. 1999.
57. P. Yu, K. Wu, K. Lin, and S. Son, “On real-time databases: Concurrency control and scheduling,” Proceedings of IEEE, Special Issue on Real-Time Systems, vol. 8, no. 1, 1994, pp. 140–157.
58. Y. Zhang, H. Kameda, and K. Shimizu, “Adaptive bidding load balancing algorithms in heterogeneous distributed systems,” in Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994.