Real-Time Processing in Client-Server Databases

Vinay Kanitkar and Alex Delis, *Member, IEEE Computer Society*

Abstract—In this paper, we propose and experimentally evaluate the use of the client-server database paradigm for real-time processing. To date, the study of transaction processing with time constraints has mostly been restricted to centralized or "single-node" systems. Recently, client-server databases have exploited locality of data accesses in real-world applications to successfully provide reduced transaction response times. Our objective is to investigate the feasibility of real-time processing in a data-shipping client-server database architecture. We compare the efficiency of the proposed architecture with that of a centralized real-time database system. We discuss transaction scheduling issues in the two architectures and propose a new policy for scheduling transactions in the client-server environment. This policy assigns higher priorities to transactions that have a greater potential for successful completion through the use of locally available data. Through a detailed performance scalability study, we investigate the effects of client data-access locality and various updating workloads on transaction completion rates. Our experimental results show that real-time client-server databases can provide significant performance gains over their centralized counterparts. These gains become evident when large numbers of clients (more than 40) are attached per server, even in the presence of high data contention.

Index Terms—Client-server databases, real-time transaction processing, experimental performance evaluation.

1 Introduction

The proliferation of network-centric computing has created opportunities for corporations and institutions to support their business cycle with large-scale information systems. Interagency transactions, known collectively as electronic commerce (e-commerce) [11], [31], have led to new models of interaction among organizations and individuals. In this environment, the movement/transfer of funds and equity is performed electronically and needs to be completed within predefined deadlines. If implemented effectively, such execution of commercial operations offers a cost-effective means of managing global financial services. Modern commercial applications often require that the duration of their transactions obeys time-constraints in varying degrees and they are based on the efficient interaction among distinct computing systems. Real-Time Systems (RTS) and Client-Server Databases (CS-DBS) are two key areas that can assist in achieving the above goal. Although there has been a great deal of independent research in these two areas, their aggregation has not been investigated at all.

In a real-time system, tasks submitted to the system for execution have deadlines imposed by the application requirements. By utilizing elegant scheduling techniques and exploiting a priori knowledge of the nature of the tasks, RTSs have been designed so that time constraints on individual jobs are met with as small a percentage of missed deadlines as possible [1], [12], [22], [29], [38]. On a separate track, client-server databases have been designed to take advantage of rapid improvements in computing power and increasing network data transfer capabilities in order to provide high throughput rates. This performance enhancement has been derived by effectively utilizing the resources available in a network of clients [7], [41], [42]. While algorithms for scheduling real-time tasks [1], [15], [18], [29], real-time multiprocessor systems [6], [30], [27], and the assignment of tasks in distributed environments [22] have been addressed, there has been no coverage of real-time issues in the popular client-server database paradigm [8], [10], [23], [40]. In this paper, we make the case for real-time transaction processing in a CS-DBS environment and provide indicators of the performance and scalability of such systems. Our objective is to establish, through experimental means, the viability and the usefulness of the proposed framework.

In real-time database systems (RTDBS), transactions—that involve nonnegligible I/O operations—are scheduled with constraints on their required completion time. Time constraints are usually specified in the form of deadlines. A deadline is the latest possible time by which a transaction must complete in order to be useful. Therefore, an important measure of the effectiveness of a RTDBS is the percentage of transactions that are completed within their specified deadlines. In this paper, we investigate the performance of a CS-DBS in the presence of real-time tasks. The aggregation of these two areas is termed Client-Server Real-Time Databases (CS-RTDBSs). In the proposed framework, we use the resources available at client sites and exploit intertransaction data caching in order to support real-time processing. A centralized server-based real-time database system (CE-RTDBS) is used as the baseline case. This baseline is used to determine the operational parameters and workloads

- *V. Kanitkar is with Akamai Technologies Inc., 500 Technology Square, 3rd Floor, Cambridge, MA 02139. E-mail: kanitkar@akamai.com.*
- *A. Delis is with the Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201. E-mail: ad@nyxos.poly.edu.*

Manuscript received 17 May 1999; revised 27 Dec. 2000; accepted 27 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110437.

under which a CS-RTDBS configuration can offer promising performance results. We chose not to use a distributed real-time database system as the baseline for two reasons: 1) Most current implementations of real-time databases are centralized and 2) in the absence of transaction-shipping between multiple database nodes, the flexibility of a distributed database system is limited.

Although client sites in this paper often seem to be directly accessible by users, that need not necessarily be the case. A formed cluster—between the client sites and their supporting server—can be used as the "nucleus" system configuration running applications on a dedicated and/or a virtual private network (VPN). Users submit their transactions/requests from the "periphery" of this nucleus. Requests are handled by client sites that demonstrate physical proximity to users and/or load balancing and sharing considerations. For example, consider a bank with multiple internetworked branches connected and operating as a client-server system. Here, employees of any one branch would only have to communicate with the client site serving their branch. Account data for customers of a bank that initiate transactions at any branch would be cached at the client site serving that branch and used for their banking needs. The proposed CS-RTDBS configuration can be used to facilitate the effective development of a wide range of application systems. This set of application software includes:

1. Highly Available Database Services: Such database systems make up the core of many telecommunication operations and their goal is to not only manage voluminous data in real-time conditions with virtually zero downtime, but also to provide customers with advanced billing options and services. Data fragmentation and a shared-nothing approach have been proposed as a way to develop such services [19]. 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 rapidly changing workload characteristics while complying with prespecified quality-of-service requirements. In this context, new strategies for data placement in video/audio applications operating in variable bit-rate and 3D interactive virtual worlds are matters of recent proposals [34].
3. Ultrafast Internet Content Delivery: There exist multiple concurrent efforts that attempt to bring web content closer to the requesting user. This can be either by using multiple proxies or content facilities around the globe and trying to always furnish "updated" content. These facilities could be implemented as clusters of sites with characteristics similar to those advocated in this paper.
4. Efficient Access for Massive E-Commerce User Communities: In seeking ways to ensure that there are available resources to service user requests at all times, prominent retailers and corporations plan and implement multiserver systems capable of isolating classes of requests [2], [3], [5], [14]. In doing so, organizations can overcome server overloads and response delays. The development of such multiserver farms can follow our system model.
5. WAP-Related Infrastructure: The Wireless Access Protocol was developed as a mechanism to enable access to information services (resident on the Internet) from a wide range of PDAs and phones with limited computing resources [13], [37]. The core of this infrastructure is the WAP proxy, whose role is to translate HTML to the WAP Markup Language (WML)—WML is fashioned to be syntactically close to XML. The immense existing user community of cellular phones and PDAs can create serious performance bottlenecks in the utilization of such WAP proxies; these can be removed by deploying proxies following our proposal in this paper.

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

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

2 RTDBS Configurations

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

2.1 Models and Assumptions

This section describes the transaction processing models of the centralized and client-server (CS) database architectures. Here, we assume that the database is a collection of uniquely identifiable objects. In the centralized real-time database architecture (Fig. 1), CE-RTDBS, the database server performs all the transaction processing. The clients in such a system are assumed to be simple terminals and serve as user-interface devices only. Clusters of clients are managed by a terminal server which handles all communication between the terminals (clients) and the centralized server. Transactions are initiated at the clients and are immediately transported to the server for execution. There, the transactions are scheduled by a single centralized scheduler according to some priority assignment algorithm. Since concurrent execution of transactions is possible, a locking protocol is used to serialize access to database objects. The results of executing the transactions are

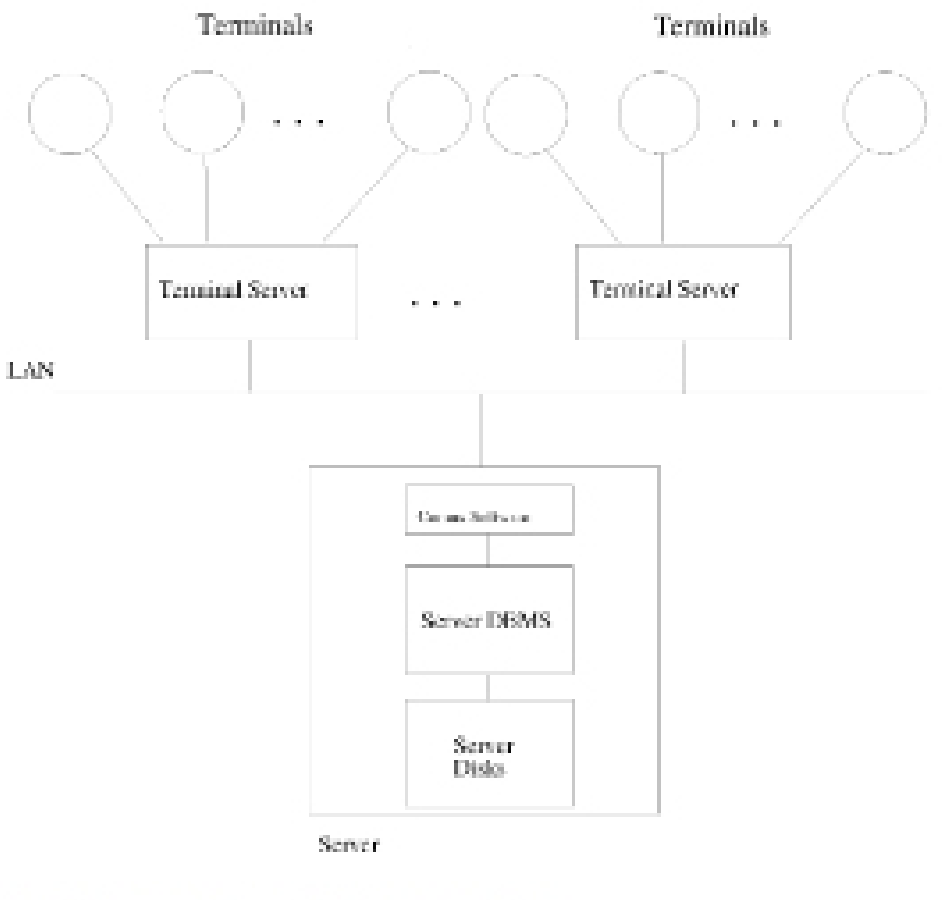

Fig. 1. The centralized database model.

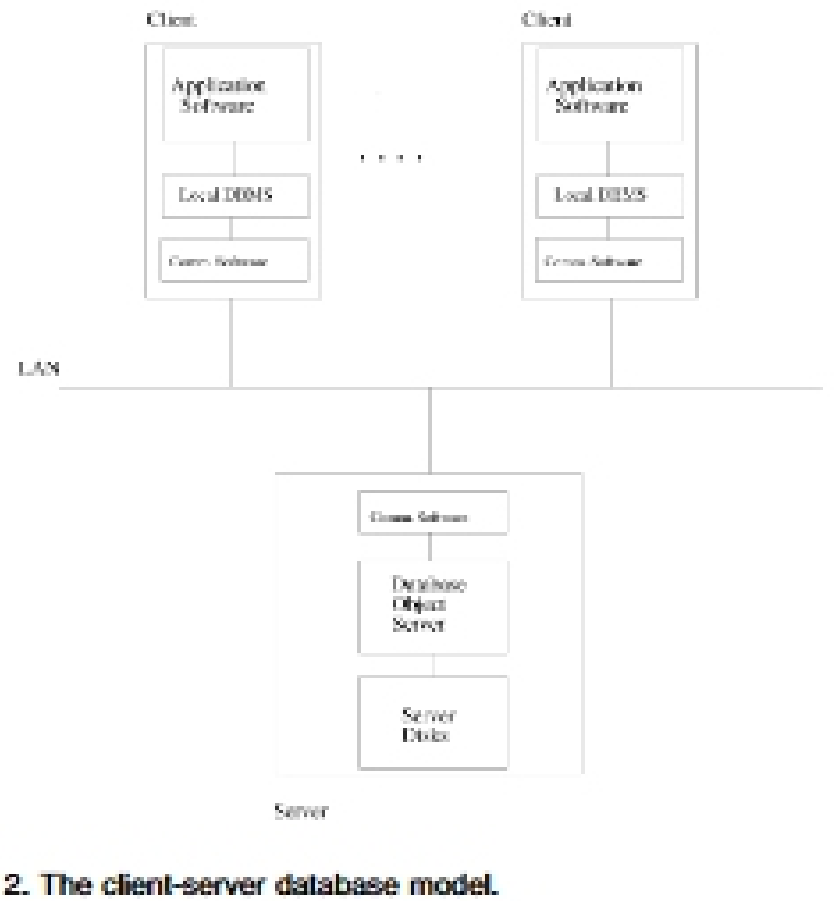

Fig. 2. The client-server database model.

communicated to users through their terminals. This is an early form of the query-shipping approach [33].

In a data-shipping CS architecture [28], [32], client sites are expected to have significant processing and storage capabilities (Fig. 2). Transactions are initiated, scheduled, and executed at client sites. The locally available buffer space and CPU of a client are used to carry out the necessary high-level database processing. This includes application-specific presentation management, query processing, and transaction management. If a database object referenced by a transaction is not cached locally, then it must first be fetched from the server before it can be used. The server ships this object to the client and the transaction that requested the object is executed locally. Thus, the server performs only low-level database functionalities (I/Os, buffering, and management of concurrency) on the behalf of requesting clients. The set of objects cached in a client's disk and memory buffers is treated as a local dataspace. Objects/locks that have been fetched from the server are returned to the server only if the latter explicitly calls them back. This type of intertransaction caching allows future requests on the cached objects/locks to be satisfied without any interaction with the server. However, intertransaction caching of data also complicates database recovery in case of client failures. The durability of committed client transactions after a client crash needs to be ensured. Although, we consider database recovery to be beyond the scope of this paper, we envision the possibility of extending an algorithm like ARIES/CSA [26] to this environment in the future.

Global concurrency control is based on pessimistic locking and is performed by the server using a lock table. Clients are permitted to hold two types of locks, shared (read-only) and exclusive (read-write) [17], [28], [41]. More than one client can hold shared locks (SL) on an object, but an exclusive lock (EL) can be granted to only one client at a time. Additionally, if a client holds an exclusive lock on an object, then no other client can have any type of lock on that object. When a lock is granted to a client, a copy of the object is shipped over to the requesting client (if necessary). If another client has a conflicting lock on that object, the server then contacts that client and requests it to release the lock on the object and return the object, if necessary. This is done by the client as soon as no local transactions are accessing the object in question. Once the object has been returned, the server grants the lock to the requesting client and dispatches the object to it.

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

In the CS-DBS, each client also has a local lock manager [25]. Since each client can execute multiple transactions concurrently, it is necessary for all transactions to acquire appropriate locks on the objects they access. Transactions are allowed to acquire shared or exclusive locks on locally cached objects, but the lock that can be granted by the local