StarBase: A Firm Real-Time Database Manager for Time-Critical Applications

Matthew R. Lehr, Young-Kuk Kim and Sang H. Son

Computer Science Department
University of Virginia
Charlottesville, VA 22903, USA

Abstract

Previous work in real-time database management systems (RT-DBMS) has primarily focused on simulation. This paper discusses how current real-time technology has been applied to architect an actual RT-DBMS on a real-time microkernel operating system. A real RT-DBMS must confront many practical issues which simulations typically ignore: race conditions, concurrency, and asynchrony. The challenge of constructing a RT-DBMS can be divided into three basic problems: dealing with resource contention, dealing with data contention, and enforcing timing constraints. In this paper, we discuss approaches to each problem.

1 Introduction

As real-time applications increase in complexity, so do their data requirements. For several years, researchers have sought a general solution to the problem of collecting, storing, and retrieving data in real time by devising database management systems to manage data in a time-cognizant and predictable manner [7]. Despite all of its features, a conventional DBMS is not quite capable of meeting the demands of a real-time system. Typically, its goals are to maximize transaction throughput, minimize response time, and provide some degree of fairness. A real-time DBMS system, however, must adopt goals which are consistent with any real-time system: providing the best service to the most critical transactions and ensuring some degree of predictability in transaction processing.

StarBase is a firm real-time DBMS which supports the concurrent execution of transactions and seeks to minimize the number of high-priority transactions that miss their deadlines. StarBase uses no *a priori* information about the transaction workload and discards tardy transactions at their deadline points. StarBase runs on top of RT-Mach, a real-time operating system under development at Carnegie Mellon University [10].

StarBase differs from previous RT-DBMS work [1, 2, 3] in that a) it relies on a real-time operating system which provides priority-based scheduling and time-based synchronization, and b) it deals explicitly with data contention and deadline handling in addition to transaction scheduling, the traditional focus of simulation studies. The design of StarBase appears in Figure 1.

The StarBase DBMS receives transaction requests from database clients and places them on a priority queue. It is assumed that database clients are physically disparate from the server, so they pass messages to communicate with the DBMS server. Transaction requests are sent via RT-Mach's Inter-Process Communication (IPC) mechanism and are queued at the server's service port. RT-Mach provides a naming service with which StarBase registers its service port during initialization. Clients look up the service port by querying the name server with StarBase's well-known name. There are a fixed number of threads (light-weight processes), called Transaction Managers (TrMgr's), which dequeue those requests and perform the basic operations which constitute the transaction. The Transaction Processing unit in turn implements these basic operations. The transaction managers rely on lower-level services to obtain the resources (memory, relations, etc.) necessary for the transaction. These services are provided by the Concurrency Controller (CCMgr), the Relation I/O Manager (RIOMgr), and the Small Memory Manager (MemMgr). Each resource manager must ensure that transactions access their resources in a consistent and orderly fashion. Transaction deadlines are enforced by special Deadline Manager (DMgr) threads.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1995**	2. REPORT TYPE	3. DATES COVERED **00-00-1995 to 00-00-1995**
4. TITLE AND SUBTITLE **StarBase: A Firm Real-Time Database Manager for Time-Critical Applications**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**6**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

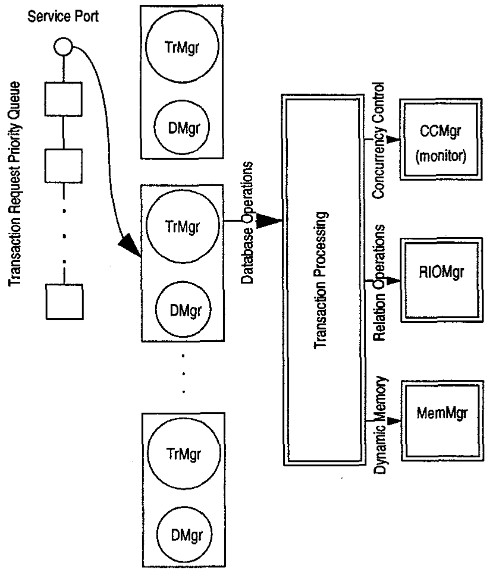

Figure 1: StarBase Server Architecture

2 Problems Unique to RT-DBMS

There are essentially three problems that real-time DBMSs must solve: resolving resource contention, resolving data contention, and enforcing timing constraints. As with other real-time systems, tasks to be performed are stratified according to their relative importance to the system. Priority combines this relative importance with task timing constraints to provide a picture of which of the many tasks should be scheduled at any given moment. The intent is to always grant the highest priority tasks access to resources (CPU, critical sections, etc.). Similarly, StarBase considers each transaction a task in its own right and seeks to provide the best service to the highest priority transactions.

3 Approaches to Resource Contention

Traditional database systems have sought to increase efficiency by sharing resources, but conventional methods for resolving which thread of execution gets which resource at any given time generally emphasize fairness, minimal response time, or maximal throughput time. As previously mentioned, the goal of real-time systems is often to minimize the number of high-priority transactions which miss their deadlines. Researchers have developed two techniques to resolve resource contention in a real-time setting. First, priority-cognizant CPU scheduling algorithms [6] such as Rate Monotonic, Earliest Deadline First, and Fixed Priority, afford the most CPU time to tasks with the highest priorities. Second, for non-preemptible resources, a protocol called Basic Priority Inheritance [9] is used to ensure the highest priority tasks access busy resources within a bounded period of time.

StarBase, like other applications, is highly depended on its native operating system, RT-Mach [10], to help it resolve resource contention. RT-Mach provides several priority-based scheduling regimes, including Fixed Priority, Earliest Deadline First, Rate Monotonic, and Deadline Monotonic. RT-Mach's real-time thread model [10] distinguishes real-time threads of execution from ordinary ones, requiring the explicit specification of timing constraints and criticality on a per-thread basis. The timing and priority information is then used as input to the RT-Mach scheduler.

StarBase uses RT-Mach's real-time message passing (RT-IPC) to ensure that transactions enter service in priority order. Once a Transaction Manager accepts a transaction request, RT-IPC ensures the Transaction Manager executes at the transaction's priority. RT-Mach then determines the rate at which the Transaction Manger (and hence the transaction) progresses according to its priority-based CPU scheduling. Finally, resource managers such as the Concurrency Controller and Small Memory Manager are guarded by real-time synchronization mechanisms (RT-Sync) to ensure the highest priority Transaction Mangers have the best access to resources.

For purposes of uniformity, StarBase adopts the same data type that RT-Mach uses to convey priorities, facilitating the straightforward translation of StarBase to RT-Mach priorities. Since the priority data type, `rt_priority_t`, includes a wide range of criticality and timing information, major changes in scheduling policy (e.g., Fixed Priority to Earliest Deadline First) are reduced to simple changes in the functions which compare priorities (e.g., changing the comparison of criticalities to one of deadlines) without any change in the client/server interface. StarBase itself must make priority-based decisions (e.g., concurrency control), so its priority-based comparisons involve priorities expressed as `rt_priority_t`-typed values. Of course, which policy is most appropriate differs from application to application, so the policy is to be used is left as a compile-time constant. Naturally, StarBase must use a consistent transaction scheduling policy across all of its priority-based decisions.

Since performance ultimately degrades as the number of threads of execution in a system increase, and lazy allocation of resources adds unpredictability to the system, StarBase maintains only a fixed number of preallocated transaction manager threads. At the same time, since the StarBase DBMS has no *a priori* knowledge of transaction workload, more transactions may be submitted to the DBMS than it can handle at any given time. In order to throttle the flow in such circumstances, StarBase needs a mechanism to decide which requests to admit into service, and RT-Mach's RT-IPC facilities do just that in a convenient and priority-cognizant manner.

To submit a transaction to the StarBase DBMS, a client places the transaction instructions and priority information into a message and uses RT-IPC to send the message to the DBMS server. Since RT-IPC queues incoming messages in priority order, the next available transaction manager receives the next highest priority unreceived message. Requests are therefore served in priority order and only the highest priority outstanding requests are in service at any given time. If a high priority transaction request cannot be serviced immediately because all transaction manager threads are busy serving some lower priority requests, RT-IPC's priority inheritance expedites one or more of the transaction managers so that the high priority request enters service at a time bounded by the minimum of the in-service transaction deadlines.

Once transactions enter service, StarBase needs to ensure that high priority transactions progress as quickly as possible. Since transactions require real-time execution, StarBase creates one real-time thread for each transaction manager and relies on RT-Mach's real-time CPU scheduling to schedule them. Transaction manager priorities are not specified explicitly by StarBase, however. Each obtains the correct priority assignment automatically upon receipt of a new transaction via RT-IPC's priority handoff mechanism [4].

Transactions, depending on the nature of their operations, require some dynamic allocation of memory during their execution. StarBase maintains a Small Memory Manager to allocate and manage dynamic memory. Since transaction managers of different priorities may attempt to use it simultaneously, entry into the Small Memory Manager is guarded by a real-time mutex variable to avoid the priority inversion problem and to ensure the heap is accessed in mutual exclusion. To provide (relatively) predictable access to memory allocated through the manager, the heap is *wired* so that it cannot be paged out of physical memory.

Although the StarBase concurrency controller is responsible for resolving contention at a higher level (i.e., data contention), it still relies on RT-Mach to provide basic synchronization and avoid the priority inversion problem. In particular, the concurrency controller must keep its own data structures consistent and ensure that transaction commits occur without interference. As such the concurrency controller is organized as a monitor, with a single real-time mutex variable for the monitor lock, and one real-time condition variable for each transaction manager.

4 Approaches to Data Contention

In addition to resource contention, StarBase faces a problem unique to DBMSs: data contention. DBMS interpose themselves between applications and the raw unstructured storage media to provide the illusion of atomic operations called transactions. In order to provide this illusion, DBMS use concurrency control algorithms to give the appearance that the data contained in relations is a result of a serial execution of these transactions. There are two major types of concurrency control which have been considered for use in real-time databases: lock-based and optimistic methods. In general, lock-based methods delay transactions to avoid having them access data in an inconsistent way, whereas optimistic methods abort transactions.

StarBase uses a real-time optimistic concurrency control method called WAIT-X [2], which has been experimentally shown to outperform lock-based concurrency control methods in a firm real-time setting. With WAIT-X a transaction, T, executes unhindered until it reaches the point where it can commit (i.e., make permanent their changes to the data) and WAIT-X determines which transactions T's execution conflict with. Unlike conventional concurrency control, WAIT-X employs a priority-cognizant commit test: If high-priority transactions comprise less than X% of all of T's conflictors, T can commit, aborting all conflictors in the process. Otherwise T waits so that higher priority transactions may proceed.

It was found experimentally that low values of X tend to minimize the deadline miss ratio for light loads, and high values of X tend to minimize the deadline miss ratio for heavy loads. When X = 50% is used as the threshold value, it minimizes the overall deadline miss ratio, but applications which require minimization of the highest-priority deadline miss ratio must use a greater value for X.

As an extension to WAIT-X, StarBase uses a special conflict-detection scheme called Precise Serialization [5]. Precise Serialization is an improvement over WAIT-X's usual conflict detection and is aimed at reducing the number of irreconcilable conflicts (and hence the number of transactions which must abort). A validating transaction which conflicts in a certain way (i.e., has write-read conflicts) is allowed to commit, but its conflictors must behave as if they cannot see the results of the validator.

Consider a case where a validator, T_V, attempts to commit and write a data item x which another uncommitted transaction T_{CR} has read but not written. A strict prospective validation checks the writeset of the validator against the readset of its potential conflictors, identifying write-read conflicts. If it detects such a conflict, the resolution requires aborting some of the conflicting transactions. Note, however, that if T_{CR} were to commit first, there would be no conflict on data item x. In Precise Serialization, it allows T_V to commit while T_{CR} is still running, but requires T_{CR} to behave as if it had committed before T_V. T_{CR} is constrained so that it cannot read any data item written by T_V because it would see a "future" value, and it cannot write any data item read by T_V since T_V has committed and cannot change the past. Finally T_{CR} must discard (as late writes) updates to any data items which T_V wrote during its commit. This pseudo-reserialization of T_V and T_{CR} is called backward ordering and its goal is to increase the probability that potential conflictors can complete without either aborting and restarting.

Since Precise Serialization is a conflict-detection scheme, not a full-blown method of concurrency control, it supplements StarBase's WAIT-X implementation rather than replacing it entirely. Precise Serialization modifies the WAIT-X validation conflict detection and requires the addition of a mechanism to detect when a pseudo-reserialized transaction does not behave in accordance with its virtual order in the execution history.

During validation, Precise Serialization partitions the set of conflicting transactions into those which conflict reconcilably and those which conflict irreconcilably. Should the validator be allowed to commit, the reconcilable conflictors must be pseudo-reserialized by backward ordering, while the irreconcilable conflictors must be aborted. To keep track of which are which, StarBase maintains a reserialization candidate set for the validator in addition to the conflict set of the WAIT-X implementation described previously. The conflict set still identifies which transactions conflict irreconcilably with the validator, but the candidate set identifies precisely those datasets among which reconcilable write-read conflicts exist.

To construct the candidate set and the conflict set at the point of validation, the CCMgr cycles through each dataset referenced by the validator, T_V. If T_V has only a write-read conflict with an uncommitted transaction, T_{CR}, on a dataset, then the serialization order should be $T_{CR} \rightarrow T_V$ (backward validation) and the conflicting datasets are added to the reserialization candidate set. If T_{CR} has only a write-read conflict with T_V, then the serialization order should be $T_V \rightarrow T_{CR}$ (forward validation). In this case T_V and T_{CR} are considered to be non-conflicting. If the CCMgr determines that the serialization order should be simultaneously $T_{CR} \rightarrow T_V$ and $T_V \rightarrow T_{CR}$, then T_V and T_{CR} are irreconcilably conflicting, and T_{CR} is added to the conflict set. Note that the CCMgr does not consider write-write conflicts since transactions are required to read tuple locations to determine their values or to establish that they are empty before writing them. Consequently a writeset is always a subset of the readset (for a given transaction and relation) and checking both against a potential conflictor's writeset is redundant.

Once the candidate set and conflict set are completely identified, the CCMgr determines whether the validator should commit or wait according to the WAIT-X commit test. If the validator waits, the conflict and candidate sets are discarded—they will be recomputed if and when the validator retries validation. If the validator commits, the transactions in the conflict set are aborted and the CCMgr must pseudo-reserialize the reconcilable conflictors. Pseudo-reserialization is achieved by attaching copies (or *remnants*) of T_V's datasets to those datasets with which they conflict–note that these dataset pairs are precisely those comprising the reserialization candidate set. Thus when a conflictor later updates its read- and writesets, it can quickly check whether the operation violates its virtual order in the execution history by consulting the dataset remnants attached to the dataset involved in the operation.

Since one of T_V's datasets may conflict with more than one of the conflictors', a remnant is given a reference count rather than physically copied. As conflictors commit or abort one by one, the CCMgr decrements the reference count. When the last conflictor terminates, the CCMgr discards T_V's dataset remnant.

5 Deadline Enforcement

Each StarBase transaction is accompanied by a deadline specification. StarBase enforces deadlines on each transaction with the aid of RT-Mach facilities. Applications submit transactions to StarBase with application-determined criticality and deadline information. Since StarBase is a firm-deadline system, it attempts to process the transaction and reply to the application at or before the deadline; no processing occurs after the deadline.

Firm deadline transactions may be contrasted with soft deadline transactions which are viewed as having some usefulness even if their execution extends beyond the deadline point. Hard deadline transactions are those transactions whose failure to execute on time is viewed as catastrophic.

RT-Mach provides the concept of a real-time *deadline handler*, a separate thread of execution which performs application-specific actions when the deadline expires. Typical actions are to abort the thread (firm deadline) or lower its priority (soft deadline). In addition to RT-Mach's real-time threads, implementation of a deadline handler requires time-based synchronization. In order to ensure the handler action is ready to execute before the deadline, the real-time deadline handler must be eagerly allocated as a real-time thread to execute the deadline handler code. The deadline handler thread then uses a *real-time timer* to block the thread until the deadline expires. A real-time timer is an RT-Mach abstraction which allows real-time threads to synchronize with particular points in time as measured by real-time clock hardware devices [8].

RT-Mach provides a default deadline handler constructed from the building blocks discussed above, but it is inadequate for StarBase's purposes. First, the default deadline handler supports only threads with uniform deadlines. StarBase, since it assumes no *a priori* information about its transaction workload, requires that its deadline handlers adapt to new transactions and their deadlines as they enter service. Secondly, a RT-Mach default deadline handler forcibly suspends a thread when it misses its deadline so that the thread does not interfere with the handler's execution. If a thread misses its deadline while in the middle of a critical section, it is suspended and cannot leave the critical section until it is resumed. StarBase uses a critical section to resolve potential race conditions between transaction commit (by the transaction manager) and deadline abort (by the deadline manager), so use of a RT-Mach-style deadline handler can result in deadlock. Thirdly, default deadline handlers do not allow the transaction and deadline managers to synchronize cooperatively. A deadline manager must know when a transaction completes so that it does not generate a useless abort; a transaction manager must know when the deadline expires, so that it does not commit the aborted transaction. Neither is possible without some shared state which must be accessed in mutual exclusion.

The solution is to devise a deadline handler implementation which handles variable deadlines, avoids potential deadlocks, and is eagerly allocated to provide some degree of predictability but at the same time takes precedence over the transaction it manages when the transaction deadline expires.

StarBase uses RT-Mach's real-time clock and timer facilities, along with the real-time thread model to provide a special deadline handler which will attempt to abort the transaction just before the deadline and reply to the client.

First, a Deadline Manager must synchronize with the Transaction Manager with which it is paired and obtain the deadline of each transaction before it enters service. Next, the Deadline Manager must wait either for the transaction to complete or for the deadline to expire. If the deadline expires, the Deadline Manager must abort the transaction asynchronously with respect to the Transaction Manager processing it. StarBase has a scheme to ensure that the Deadline Manager is scheduled in preference to its Transaction Manager by assigning it a slightly higher criticality and slightly tighter timing constraints than the Transaction Manger.

6 Conclusions and Future Work

In this paper, we have presented the architecture to support a firm real-time DBMS assuming no *a priori* knowledge of transaction workload characteristics. Unlike previous simulation studies, StarBase uses a real-time operating system to provide basic real-time functionality and deals with issues beyond transaction scheduling: resource contention, data contention, and enforcing deadlines. Issues of resource contention are dealt with by employing priority-based CPU and resource scheduling provided by the underlying real-time operating system. Issues of data contention are dealt with by use of a priority-cognizant concurrency control algorithm with a special conflict-detection scheme, called Precise Serialization, to reduce the number of aborts. Issues of deadline-handling are dealt with by constructing deadline handlers which synchronize with the start and end of a transaction and which don't interfere with its execution until the deadline expires.

The correctness of the concurrent execution of transactions on StarBase has been demonstrated by a special demonstration program. The program graphically depicts a high-priority writer obtains better performance than a concurrently-executing low-priority writing transaction. There is still plenty to accomplish, however. Future work includes the extension of the query language to include set-based relational algebra operations, the reduction of data conflicts by the use of a specially-designed indexing mechanism. Another interesting future work is to extend these solutions to the situation in which transaction characteristics are at least partially specified beforehand. With prior knowledge, a real-time DBMS can preallocate resources and arrange transaction schedules to minimize conflicts, resulting in more predictable service. Execution time estimates and off-line analysis can be used to increase DBMS-wide predictability. We also plan to port StarBase to other real-time operating systems and identify operating system services essential to supporting StarBase. Once the basic, real-time, POSIX.4-compliant functionality needed to support a firm real-time database has been established, StarBase can be ported to other platforms.

Acknowledgments

This work was supported in part by Loral Federal Systems, VCIT, and ONR.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] J. R. Haritsa. *Transaction Scheduling in Firm Real-Time Database Systems.* PhD thesis, University of Wisconsin–Madison, August 1991.

[3] J. Huang. *Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation.* PhD thesis, University of Massachusetts at Amherst, May 1991.

[4] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC Extension for Real-Time Mach. Technical report, Carnegie-Mellon University, August 1993.

[5] J. Lee and S. H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 66–75, Raleigh-Durham, NC, December 1993.

[6] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[7] Krithi Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(1), 1992.

[8] S. Savage and H. Tokuda. Real-Time Mach Timers: Exporting Time to the User. In *Proceedings of the Third USENIX Mach Symposium*, April 1993.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[10] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.