ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

A Study on Real Time Database System for Distributed Applications

Poonam B. Lohiya[1], S. G. Pundkar[2], D. H. Deshmukh[3], K. H. Deshmukh[4]

Assistant Professor, Department of Computer Engineering, PRMIT&R, Badnera, India[1,2,3,4]

ABSTRACT: A real-time database system (RTDBS) is a database system providing all features on traditional database system such as data independence and concurrency control, while at the same time enforces real-time constraints that applications may have. Like a traditional database system, a RTDBS functions as a repository of data, provides efficient storage, and performs retrieval and manipulation of information. However, as a part of a real-time system, tasks have time constraints; a RTDBS has the added requirement to ensure some degree of confidence in meeting the system's timing requirements. A real-time database is a database in which transactions have deadlines or timing constraints. Real-time databases are commonly used in real-time computing applications that require timely access to data. And, usually, the definition of timeliness is not quantified; for some applications it is milliseconds, and for others it is minutes. In this paper we have discussed the issues related with timing constraint for distributed real time application.

I. INTRODUCTION

In general, data in a real-time system is managed on individual basis by every task within the system. However, with the advancement of technology, many applications are requiring large amounts of information to be handled and managed in a timely manner. Thus, a substantial number of real-time applications are becoming more data-intensive. Such larger amounts of information had produced an interdependency relationship among real-time applications. Therefore, in various application domains, data can no longer be treated and managed on individual basis, rather it is becoming a vital resource requiring an efficient data management mechanism. Meanwhile, database management systems are designed around such a concept; that is, with the sole goal of managing data as a resource. Hence, the principles and techniques of transaction management in Database Management Systems need to be applied to real-time applications for efficient storage and manipulation of information. In an attempt to achieve the advantages of both systems, real-time and database, continuous efforts are directed towards the integration of the two technologies. Such an integration of the two technologies resulted in combined systems known as *Real-Time Database Systems*. Today, many applications require such systems, i.e., information retrieval systems, airline reservation systems, stock market, banking, aircraft and spacecraft control, robotics, factory automation, and computer-integrated manufacturing, and the list is vast. Traditional database systems differ from a RTDBS in many aspects. Most important RTDBSs have different performance goal, correctness criteria, and assumptions about the applications. Unlike a traditional database system a RTDBS may be evaluated based on how often transactions miss they deadlines, the average lateness or tardiness of late transactions, the cost incurred in transactions missing their deadlines, data external consistency and data temporal consistency. Numerous real-word applications contain time-constrained access to data as well as access to data that has temporal validity. Consider for example a telephone switching system, network management, navigation systems, stock trading, and command and control systems. Moreover consider the following tasks within these environments: looking up the obstacle detection and avoidance, radar tracking and recognition of objects. All of these entail gathering data from the environment, processing information in the context of information obtained in the past, and contributing timely response. Another characteristic of these examples is that they entail processing both temporal data, which loses its validity after a certain time intervals, as well as historical data. One of the most important points to remember here is that real-time does not just mean fast [9]. Furthermore, real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage explicit time constraints in a predictable fashion, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks. Databases are useful in real-time applications because they combine several features that facilitate

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

(1) The description of data,
(2) The maintenance of correctness and integrity of the data,
(3) Efficient access to the data, and
(4) The correct executions of query and transaction execution in spite of concurrency and failures.
Real time database applications are normally structured in to transactions. A transaction is a sequence of reads and writes on the database to achieve some high-level function of the application [1][2][8].

II. TYPES OF REAL-TIME TRANSACTIONS

- **Hard real-time transactions**

Many applications exist in which hard real-time transactions must be supported; examples include embedded control applications, defense systems, and avionics systems. With the recent advent of workstation-class multiprocessor systems, multiprocessor-based implementations of these systems are of growing importance. Un- fortunately, the problem of implementing hard real-time transactions on multiprocessors has received relatively little attention. With most (if not all) previously-proposed schemes for implementing such transactions, the kind of transactions that can be supported is often limited, and worst-case performance is often poor, adversely impacting schedulability. Transactions have many attributes (priority, execution time, resource usage, etc.), the most important of which is the deadline. A transaction's deadline is the mean by which the system designer can specify its timeliness requirement. A way of classifying transactions is based on the effects of their deadlines not being met. Hard real-time transactions are those which can have catastrophic consequences if they are not executed on time. Soft real-time transactions, on the other hand, will only degrade the system's performance when not completed within their deadline. A special type of soft real-time transaction is a firm real- time transaction, which has no value to the system when its deadline is not met and should therefore be aborted. A pragmatic way to design real-time databases is to minimize the ratio of transactions that miss their deadlines. This strategy is, however, a step short of meeting hard real-time requirements, under which missing a single deadline can lead to catastrophic effects. Most commercial databases that claim real-time properties are not suitable for hard real-time applications. Real-time database designers are faced with a vast number of constraints, difficult to fulfill while offering an adequate set of features. The database source code, developed to manage the transactional mechanisms, has to be thoroughly analyzed in order to guarantee worst case execution timings. Furthermore, a generic hard real-time database would have to suit a great diverseness of application requirements, thereby demanding a significant development effort. Providing hard real-time transactions limits the broadness of the set of features a database can offer. A particular method to join two tables may have an order of complexity which makes its execution depend on the number of rows in the table or require too much memory. If there is no other algorithm capable of providing the same functionality, with a feasible amount of computation, then it may be difficult to make it part of a hard real-time database. However, many different research efforts have been presented in the past years which may lead to functional hard real-time databases. At the moment there are many commercially available databases which are time-cognizant and feature a great deal of predictability. Such databases already solve to some extent the issues debated in this essay. They should be expected to evolve with the goal of performing hard real-time ACID transactions.

- **Soft real-time transactions**

In such systems, it is very difficult to guarantee that all deadlines will be met, and hence one tries to minimize the number of deadlines that are missed. Guaranteeing all deadlines may be hard because it is impossible to impractical to place an upper bound on the load. In our radar example, for instance, one may guess the maximum number of objects that may suddenly appear, but there is no way of knowing for sure that this maximum will never be exceeded. Another reason why soft real-time systems may miss deadlines is because the tasks they run are complex and it is hard to predict exactly how long they will run or what resources they may need. For instance, it is hard to know in advance how many rules the expert system in our example will trigger. In the database component, the running time of a search will depend on what other transactions are concurrently running and holding locks. Furthermore, disk access times will depend on where the previous request left the disk arm, again hard to determine in advance.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

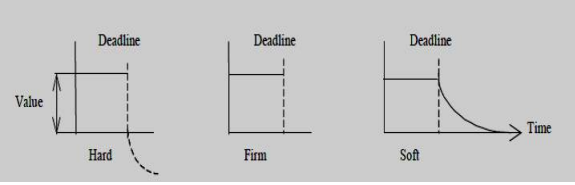

Figure1. Types of Transaction

III. SCHEDULING DISTRIBUTED REAL TIME APPLICATIONS

A scheduler in general is an algorithm or a policy for ordering the execution of the outstanding processes (tasks) on a processor according to some predefined criteria. Each task within a real-time system has a deadline, an arrival time, and possibly an estimated worst-case execution time. A task's execution time can be derived from the time each resource is required and the precedence constraints among sub-tasks. Execution time information can be given in terms of *deterministic*, *worst-case*, or *probabilistic* estimates. The responsibility of a real-time system scheduler is to determine an order of execution of the tasks that are *feasible*4. Typically, a scheduler is *optimal* if it can schedule all task sets that other schedulers can [12][13]. A scheduler may be *preemptive* or *non-preemptive*. A preemptive scheduler can arbitrarily suspend and resume the execution of a task without affecting its behavior. Preemption is used to control priority-driven scheduling. Typically, preemption occurs when a higher-priority task becomes run able while a lower-priority task is executing. On the other hand, in a non-preemptive scheduler, a task must run without interruption until completion [10][15][16]. Simulation studies in [13] showed that the use of preemption is more appropriate for scheduling real-time systems. Finally, a *hybrid* approach is a preemptive scheduler, but preemption is only allowed at certain points within the code of each task. Real-time scheduling algorithms can be classified as either *static* or *dynamic*. A static approach also known as a *fixed-priority*, where priorities are computed off-line, assigned to each task, and maintained unaltered during the entire lifetime of the task and the system. A static scheduler requires complete prior knowledge of the real-time environment in which it is deployed. A table is generated that contains all the scheduling decisions during run-time. Therefore, it requires little run-time overhead. Aside from the many disadvantages of static scheduling [NAT92], it is rather inflexible, because the scheme is workable only if all the tasks are effectively periodic. Jensen et al. stated that fixed priority scheduling could work only for relatively simple systems, and results in a real-time system that is extremely fragile in the presence of changing requirements. It was shown in that fixed-priority schedulers perform inconsistently, particularly as the load increases. On the other hand, dynamic scheduling techniques assume unpredictable task-arrival times and attempt to schedule tasks dynamically upon arrival. That is, a dynamic scheduling algorithm dynamically computes and assigns a priority value to each task, which can change at run-time. The decisions are based on both task characteristics and the current state of the system, thereby furnishing a more flexible schedule that can deal with unpredictable events. In the rest of this section, we briefly discuss the various methods used in constructing different priority driven schedulers for real-time systems.

A Few Basic Concepts

Before focusing on the different classes of schedulers more closely, let us first introduce a few important concepts and terminologies which would be used in our later discussions.

- **Valid Schedule:** A valid schedule for a set of tasks is one where at most one task is assigned to a processor at a time, no task is scheduled before its arrival time, and the precedence and resource constraints of all tasks are satisfied.
- **Feasible Schedule:** A valid schedule is called a feasible schedule, only if all tasks meet their respective time constraints in the schedule.
- **Proficient Scheduler:** A task scheduler sch1 is said to be more proficient than another scheduler sch2, if sch1 can feasibly schedule all task sets that sch2 can feasibly schedule, but not vice versa. That is, sch1 can feasibly schedule all task sets that sch2 can, but there exists at least one task set that sch2 cannot feasibly schedule, whereas sch1 can.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

If sch1 can feasibly schedule all task sets that sch2 can feasibly schedule and vice versa, then sch1 and sch2 are called equally proficient schedulers.

- **Optimal Scheduler:** A real-time task scheduler is called optimal, if it can feasibly schedule any task set that can be feasibly scheduled by any other scheduler. In other words, it would not be possible to find a more proficient scheduling algorithm than an optimal scheduler. If an optimal scheduler cannot schedule some task set, then no other scheduler should be able to produce a feasible schedule for that task set.
- **Scheduling Points:** The scheduling points of a scheduler are the points on time line at which the scheduler makes decisions regarding which task is to be run next. It is important to note that a task scheduler does not need to run continuously, it is activated by the operating system only at the scheduling points to make the scheduling decision as to which task to be run next. In a clock-driven scheduler, the scheduling points are defined at the time instants marked by interrupts generated by a periodic timer. The scheduling points in an event-driven scheduler are determined by occurrence of certain events
- **Preemptive Scheduler:** A preemptive scheduler is one which when a higher priority task arrives, suspends any lower priority task that may be executing and takes up the higher priority task for execution. Thus, in a preemptive scheduler, it cannot be the case that a higher priority task is ready and waiting for execution, and the lower priority task is executing. A preempted lower priority task can resume its execution only when no higher priority task is ready.
- **Utilization:** The processor utilization (or simply utilization) of a task is the average time for which it executes per unit time interval. In notations: for a periodic task T_i, the utilization $u_i = e_i/p_i$, where e_i is the execution time and p_i is the period of T_i. For a set of periodic tasks {Ti}: the total utilization due to all tasks $U = \sum_{i=1}^{n} e_i/p_i$. It is the objective of any good scheduling algorithm to feasibly schedule even those task sets that have very high utilization, i.e. utilization approaching 1. Of course, on a uniprocessor it is not possible to schedule task sets having utilization more than 1.
- **Jitter:** Jitter is the deviation of a periodic task from its strict periodic behavior. The arrival time jitter is the deviation of the task from arriving at the precise periodic time of arrival. It may be caused by imprecise clocks, or other factors such as network congestions. Similarly, completion time jitter is the deviation of the completion of a task from precise periodic points. The completion time jitter may be caused by the specific scheduling algorithm employed which takes up a task for scheduling as per convenience and the load at an instant, rather than scheduling at some strict time instants. Jitters are undesirable for some applications.

IV. REAL TIME SCHEDULING ALGORITHMS

The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points.

Several schemes of classification of real-time task scheduling algorithms exist. A popular scheme classifies the real-time task scheduling algorithms based on how the scheduling points are defined. The three main types of schedulers according to this classification scheme are: clock-driven, event-driven, and hybrid.

A few important members of each of these three broad classes of scheduling algorithms are the following:

1. Clock Driven
   - Table-driven
   - Cyclic
2. Event Driven
   - Simple priority-based
   - Rate Monotonic Analysis (RMA)
   - Earliest Deadline First (EDF)
3. Hybrid
   - Round-robin

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

Important examples of event-driven schedulers are Earliest Deadline First (EDF) and Rate Monotonic Analysis (RMA). Event-driven schedulers are more sophisticated than clock-driven schedulers and usually are more proficient and flexible than clock-driven schedulers. These are more proficient because they can feasibly schedule some task sets which clock-driven schedulers cannot. These are more flexible because they can feasibly schedule sporadic and aperiodic tasks in addition to periodic tasks, whereas clock-driven schedulers can satisfactorily handle only periodic tasks. Event-driven scheduling of real-time tasks in a uniprocessor environment was a subject of intense research during early 1970's, leading to publication of a large number of research results. Out of the large number of research results that were published, the following two popular algorithms are the essence of all those results: Earliest Deadline First (EDF), and Rate Monotonic Analysis (RMA). If we understand these two schedulers well, we would get a good grip on real-time task scheduling on uniprocessor. Several variations to these two basic algorithms exist.

Another classification of real-time task scheduling algorithms can be made based upon the type of task acceptance test that a scheduler carries out before it takes up a task for scheduling. The acceptance test is used to decide whether a newly arrived task would at all be taken up for scheduling or be rejected. Based on the task acceptance test used, there are two broad categories of task schedulers:

- Planning-based
- Best effort

In planning-based schedulers, when a task arrives the scheduler first determines whether the task can meet its dead-lines, if it is taken up for execution. If not, it is rejected. If the task can meet its deadline and does not cause other already scheduled tasks to miss their respective deadlines, then the task is accepted for scheduling. Otherwise, it is rejected. In best effort schedulers, no acceptance test is applied. All tasks that arrive are taken up for scheduling and best effort is made to meet its deadlines. But, no guarantee is given as to whether a task's deadline would be met.

A third type of classification of real-time tasks is based on the target platform on which the tasks are to be run. The different classes of scheduling algorithms according to this scheme are:

- Uniprocessor
- Multiprocessor
- Distributed

Uniprocessor scheduling algorithms are possibly the simplest of the three classes of algorithms. In contrast to uniprocessor algorithms, in multiprocessor and distributed scheduling algorithms first a decision has to be made regarding which task needs to run on which processor and then these tasks are scheduled. In contrast to multiprocessors, the processors in a distributed system do not possess shared memory. Also in contrast to multiprocessors, there is no global up-to-date state information available in distributed systems. This makes uniprocessor scheduling algorithms that assume central state information of all tasks and processors to exist unsuitable for use in distributed systems. Further in distributed systems, the communication among tasks is through message passing. Communication through message passing is costly. This means that a scheduling algorithm should not incur too much communication over- head. So carefully designed distributed algorithms are normally considered suitable for use in a distributed system[1,12,13,14].

V. SCHEDULING RTDB TRANSACTIONS

The primary performance determinant in a RTDB system is the policy used for scheduling transactions. A scheduling policy determines when service is provided to a transaction, thereby directly impacting whether a transaction meets its deadline. A special feature of RTDB systems, in addition to standard physical resources, is the data objects stored in the database, and transactions accessing this data have to be scheduled in accordance with real-time performance objectives. The scheduling process of transactions in a RTDB system consists of concurrency control and conflict resolution and indirectly involves recovery.

The Concept of Transactions and Serializability

The state of a database consists of records, assertions about the values of these records, and an allowed set of transformations applicable to the values of such records. These assertions are called consistency constraints. One may need to temporarily violate the consistency of the system-state while modifying it. Therefore, in order to transform a

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

consistent system-state to another consistent state, the system provides sets of actions in the form of read and writes operations. A transaction is a collection of such actions, which comprise a consistent transformation of the system-state. Each transaction, when executed alone, transforms a consistent state into a new consistent state; that is, transactions preserve consistency of the database information interleaving transactions access to the database can maximize throughput and resource utilization. Therefore, various actions of different transactions need to be executed with maximal concurrency by interleaving actions from several transactions while continuing to give each transaction a consistent view of the database. A particular sequencing of the actions from different transactions is called a schedule. A schedule that gives each transaction a consistent view of the database-state is called a consistent schedule [6]. However, failures and concurrency are the two sources of potential errors that can lead to database inconsistencies. Traditional database management systems (DBMS) prevent such inconsistencies by satisfying four properties associated with transactions, known as the ACID properties

A Atomicity: Either all or none of the transactions operations are/is performed. All the operations of a transaction are treated as a single, indivisible, atomic unit.

C Consistency: A transaction maintains the integrity constraints on the database.

I Isolation: Transactions can execute concurrently but with no interference with each other's operations.

D Durability: All changes made by a committed transaction become permanent in the database, surviving any subsequent failures. While each transaction preserves the consistency of the database at its boundaries, recovery protocols are used to ensure the atomicity and durability properties. Finally, isolation can be insured by concurrency control protocols

Conflict Resolution

While priority assignment governs CPU scheduling, conflict resolution protocols determine which of the conflicting transactions will actually obtain access to a data item. Conflicts are usually the result of concurrent executions of transactions performing incompatible operations; i.e., read vs. write on the same data item at the same time. Schedulers differ in detecting resource conflicts among transactions and the manner in which conflicts are resolved once they are detected. For shared database resources, the preempted transaction usually must be rolled back if in-place updates are being employed [18]. When a transaction requests a lock on a data item while the lock is being held, in a conflicting mode, by another transaction, both transactions have time constraints, yet only one can hold the lock. The conflict should be resolved according to the characteristics of the conflicting transactions.

VI. CONCLUSION

Real-time database systems are emerging as a structured and systematic approach to manage data, avoiding application dependent designs. This approach combines techniques from the database systems community with the existing methods for real-time computing. At least part of the database must be placed in main memory, in order to avoid the non- determinism of disks. However, a fast database executing in main memory is not necessarily a real-time database. For some applications completing each transaction within its deadline is more important than the average transaction rate. This requires a specialized design which ensures that transactions meet their deadlines. Designing a hard real-time database is not a simple task, given the vast number of constraints and the diverseness of application requirements. Real-time databases which strive to provide timeliness in addition to the well-known ACID properties are often required to sacrifice one or more such properties (e.g. timeliness vs. durability). A common strategy is to minimize the ratio of transactions that miss their deadlines. At the moment there are many commercially available databases which are time-cognizant and feature a great deal of predictability. Most of these databases are not yet suitable for hard real-time applications. Nonetheless, the numerous research efforts currently ongoing may lead to functional hard real-time databases in the near future.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 5, Issue 6, June 2016

REFERENCES

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. ACM Transactions on Database Systems, 17(3):513–560, Sept. 1992.
[2] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. Real-Time Systems, 2(4):325–346, 1990.
[3] K.-R. Choi and K.-C. Kim. T*-tree: a main memory database index structure for real time applications. In RTCSA, pages 81–88. IEEE Computer Society, 1996.
[4] C. Devor and C. R. Carlson. Structural locking mechanisms and their effect on database management system performance. Information Systems, 7(4):345–358, 1982.
[5] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In Proceedings of the Real-Time Systems Symposium, pages 94–103, Lake Buena Vista, Florida, USA, Dec. 1990.
[6] J. R. Haritsa and S. Seshadri. Real-time index concurrency control. In Real-Time Database Systems, pages 59–74, 2001.
[7] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In Proceedings of the 17th Conference on Very Large Databases, pages 35–46, Sept. 1991.
[8] H. F. Korth and A. Silberschatz. Database System Concepts. McGraw Hill, New York, NY, 1991.
[9] C. M. Krishna and K. G. Shin. Real-Time Systems. McGraw-Hill International Editions, 1997.
[10] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto, Aug. 1986.
[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM, 1(20):44–61, Jan. 1973.
[12] J. Anderson and M. Moir, \Universal Constructions for Multi-Object Operations", Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, 1995, pp. 184-193.
[13] J. Anderson and M. Moir, \Universal Constructions for Large Objects", Proceedings of the Ninth International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 972, Springer-Verlag, September 1995, pp. 168-182.
[14] J. Anderson, S. Ramamurthy, M. Moir, and K. Je_ay, \Lock-Free Transactions for Real-Time Systems", Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications, March 1996, pp. 107-114.
[15] H. Attiya and E. Dagan, \Universal Operations: Unary versus Binary", Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, 1996, pp. 223-232.
[16] B. Bershad, \Practical Considerations for Non-Blocking Concurrent Objects", Proceedings of the 13th international Conference on Distributed Computing Systems, May 1993, pp. pages 264-274.
[17] R. Bettati, End-to-End Scheduling to Meet Deadlines in Distributed Systems, Ph.D. Thesis, Computer Science Department, University of Illinois at Urbana-Champaign, March 1994. 19
[18] M. Greenwald and D. Cheriton, \The Synergy between Non-blocking Synchronization and Operating System Structure", Proceedings of the USENIX Association Second Symposium on Operating Systems Design and Implementation, 1996, pp. 123-136.