On Real–Time Transactions*

John A. Stankovic
Department of Computer and Information Science
University of Massachusetts

Wei Zhao
Department of Mathematics
Amherst College

November 1987

Abstract

Next generation real–time systems will require greater *flexibility* and *predictability* than is commonly found in today's systems. These future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications. The complexity of such systems due to timing constraints, concurrency, and distribution is high. It is accepted that the synchronization, failure atomicity, and permanence properties of transactions aid in the development of distributed systems. However, little work has been done in exploiting transactions in a real–time context. We have been attempting to categorize real–time data into classes depending on their time, synchronization, atomicity, and permanence properties. Then, using the semantics of the data and the applications, we are developing special, tailored, real–time transactions that only supply the minimal properties necessary for that class. This reduces the system overhead in supporting access to various types of data. The eventual goal is to verify that timing requirements can be met.

*This work was supported by ONR under contract NOOO14-84-K0734 and NSF under grants DCR-8500332 and SDB-8418216.

1 Introduction

Recently, there has been an increased interest in real-time systems and such systems are becoming more and more sophisticated. It is sometimes useful to categorize tasks (and/or transactions)[1] of real-time systems into hard and soft real-time. While there is no completely accepted definition for hard and soft real-time, we define *hard real-time* transactions as those transactions for which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Further, if these real-time constraints are not met there may potentially be catastrophic consequences. For such transactions it is necessary to guarantee that timing constraints are met. In contrast, while soft real-time transactions have timing constraints, there may still be some value for completing the transaction after its deadline, and catastrophic consequences do not result if these transactions miss their deadline. Soft real-time transactions are scheduled taking into account their timing requirements, but they are not guaranteed to make their deadlines.

Examples of real-time systems that contain both types of transactions are command and control systems, nuclear power plants, process control systems, flight control systems, and the space shuttle avionics system. In the future, real-time systems are expected to become more and more complex, have long lifetimes, have multiple levels of timing constraints, and exhibit very dynamic, adaptive and even intelligent behavior.

In this short paper, we put forth the hypotheses that the need for real-time transactions, as defined in the context of database systems, is largely limited to soft real-time, and that the current technology is incapable of solutions for hard real-time (except for trivial cases). We also suggest that much of the data in real-time systems does not require full transaction support, so that special mechanisms can be developed to support real-time data. Our informal definition of *real-time data* is any data that must be manipulated in some way (read, written, inserted, deleted, etc.) subject to a timing constraint. Three other contributions of this paper include two real-time access control protocols and a real-time stream primitive. The state of the art in real-time transactions is primitive. This paper presents some preliminary ideas which are being actively persued by members of the Spring project at the Univ. of Massachusetts.

[1]In real-time systems the entity with an associated time constraint is usually called a task. Since this paper is discussing transactions, we assume that transactions have associated time constraints. Consequently, we use the term transaction throughout the paper.

The remainder of the paper is organized as follows. In section 2 we briefly describe the state of the art in hard real–time computing. The purpose of presenting this material is to show the difficulty of attempting to perform hard real–time transactions with today's technology. We also argue that there seems to be limited need for hard real–time transactions, but a significant need for soft real–time transactions. Since the state of the art supports neither type, we are focussing our initial efforts on the simpler and more widely needed type – soft real–time transactions. Section 3 then proposes a virtual clock access control protocol and a value function protocol based on a pairwise comparison. Both of these protocols are for soft real–time transactions. Section 4 then describes a specific example of a hard real–time primitive called a real–time stream. Section 5 briefly summarizes the paper.

2 Hard Real–Time

One of the most critical parts of supporting real–time systems is the ability to deal with real-time constraints. Because of the large number of combinations of transactions that might be active at the same time and because of the continually changing demands on the system, it will generally be impossible to pre-calculate all possible schedules *off-line* to statically guarantee real-time constraints.

Since it is infeasible to perform static scheduling of real–time transactions, let us discuss the possibility of dynamically guaranteeing deadlines for transactions. Our hypothesis is that it is infeasible to perform an absolute guarantee of deadlines for real–time transactions where transactions are defined as providing the properties of serializability, atomicity, and permanence. It is infeasible due to two primary factors. One, it is extremely difficult, if not impossible, to determine the worst case computation time of a transaction, and even when possible, the computation time requirements often have a very large variance. To guarantee such transactions requires an enormous excess of resource capacity to account for the worst case of any possible combination of transactions being active at the same time. Two, transactions interact with the operating system and I/O subsystem in unpredictable ways. Since transactions require concurrency control, commit protocols, recovery protocols, buffers, and access to disks, using today's operating systems it is virtually impossible to predict the response time for a transaction, and that response time is usually long due to these complicated protocols and disk access times. Unpredictability, is due to operating system features such as paging, working sets, dynamic adjusting of priorities, disk scheduling algorithms, buffering schemes, and blocking over resource contention.

Since both the static and dynamic guarantees for hard real–time transactions are infeasible, what can be done? We can develop specific transaction mechanisms that explicitly account for deadlines and use these mechanisms to make a best effort at achieving deadlines. In section 3 we propose such an approach for an access control mechanism. Another thing that can be done is to develop better underlying support for real–time transactions. We now consider this latter issue in more depth.

A major research question facing the development of real–time transactions is what should be the model of the underlying operating system. To answer this, first consider current real–time systems. Real-time systems usually include a real-time kernel. However, most existing real-time kernels are simply stripped down and optimized versions of timesharing operating systems. More specifically, the general characteristics of most *current* real-time kernels typically include:

- a fast context switch,
- a small size (with its associated minimal functionality),
- the ability to respond to external interrupts quickly,
- multi-tasking with task coordination being supported by features such as ports, events, signals, and semaphores,
- fixed or variable sized partitions for memory management (no virtual memory),
- the ability to lock entities in memory,
- the presence of special sequential files that can accumulate data at a fast rate,
- priority scheduling,
- the minimization of intervals during which interrupts are disabled,
- support of a real-time clock,
- primitives to delay tasks for a fixed amount of time and to pause/resume tasks, and
- special alarms and timeouts.

These features provide a basis for a good set of primitives upon which to build real-time systems. These features are also designed to be fast which is a laudable goal. However, fast is a relative term and not sufficient when dealing with real-time constraints. The main problems with these primitives are that they do not *explicitly* address real-time constraints, nor does their use (without extensive simulations) provide system designers with a high degree of confidence that the system will indeed meet its real-time constraints. Even though such kernels are successfully used in today's real-time embedded systems, it is only at extremely high cost and inflexibility. For example, when using the above primitives it is difficult to *predict* how tasks invoked dynamically interact with other active tasks, where blocking over resources will occur, and what the subsequent effect of this interaction and blocking is on the timing constraints of all the tasks. The current technology burdens the designer with the unenviable task of mapping a set of specified real-time constraints into a priority order in such a manner that all tasks will meet their deadlines. It is common practice to attempt to verify real-time constraints under such conditions by extensive and costly simulations and testing on the actual system. One round of changes is subject to another *extensive* round of testing. As the next generation hard real-time systems become more sophisticated, it will be necessary to develop cheaper ways to guarantee real-time constraints and to meet the flexibility requirements.

The main characteristics of *next* generation hard real-time systems are:

- new operating system and task designs to support *predictability*,
- a high degree of adaptability (short term and long term),
- physical distribution (of multiprocessors) with a high degree of cooperation,
- incorporation of integrated solutions to deal with real-time, fault tolerance, and large system requirements,
- an interface to AI programs, and
- the ability to handle complex applications.

New operating system models and scheduling algorithms are needed to enhance predictability. For example, we have developed a new operating systems/scheduling algorithm paradigm [5,6,8] where conflicts over resources are *avoided* thereby eliminating the random nature of waiting for resources found in timesharing operating systems (this same feature has many other advantages including minimizing context switches since tasks are not being context switched to wait for resources). Other work such as [2,3] also deals with resources, but in a much more static manner. Even using models such as these, it would only enhance the ability to verify time constraints, not solve all the problems. Much more work is needed in this area.

As stated above, current transaction support systems do not explicitly take timing constraints into account. We need to develop soft real-time transactions. At a minimum, soft real-time transaction mechanisms need to support time constrained scheduling, forward error recovery, and a means for using semantic information. Semantic information is normally available *a priori* in most real-time application. It is also necessary to consider the criticalness [1] of transactions as a separate issue from deadline. For example, when all transactions can meet their deadline, then criticalness is not an issue. When some transaction does not meet its deadline, we wish it to be the least critical one. Since access to disks are slow, recovery by logging to disks may have to take other forms. Using semantic information about a certain data item, a soft real-time transaction might wait for time delta t, and then, if the data item is still not available, take an old value of the data rather than wait any longer.

3 Access Control for Soft Real-Time Transactions

In this section, we propose two new access control protocols which directly take into account the time constraints of transactions and, hence, are expected to have a better performance than a traditional protocol in a real-time database system. The first protocol is based on the notion of a *virtual clock* and the second on a pairwise comparison of value functions.

3.1 Overview of the Problem

A typical database transaction is a sequence of operations performed on a database. Normally, serializability, atomicity and permanence are properties supported by the transaction mechanism. In a real-time context, now consider that some of the transactions have soft real-time constraints. Such transactions should finish their execution by a deadline but still maintain serializability, atomicity, and permanence. If a real-time transaction cannot be completed by its deadline, we say it is *lost*. In reality some of the lost transactions may continue to execute and complete at a later time. However, their value is highly diminished in this case. Consequently, one important metric is the transaction loss. Obviously, a good access control protocol should minimize the transaction loss. Given that some transactions might be lost, then another metric, the criticalness of transactions, should be used to determine which transactions are lost.

Another important metric in a real-time database system is *the lateness of transaction loss* which is a metric for lost transactions and is defined as the sum of the transaction finish times minus their deadlines, where the sum is taken over all lost transactions. Any access control protocol for soft real–time transactions should minimize the lateness of transaction loss.

Minimizing transaction loss and reducing the lateness in a real-time database system is an interesting but difficult problem. Consider an example where a real-time transaction with a large deadline has locked a data item while another real-time transaction with a small deadline then requests the lock. Intuitively, from the view point of minimizing the transaction loss, it seems that we should permit the transaction with the smaller deadline to preempt[2] the transaction with a larger deadline. However, the preemption implies not only that the work done on the data by the larger deadline transaction must be undone (may simply be done by deleting a workspace copy of the data), but also that later on (when the smaller deadline transaction completes its access to the data item), the preempted transaction, if restarted, must redo the work. The decision to restart or not would be made as part of an error handler for this transaction. The resultant delay and the wasted execution time may cause one or both of these transactions, as well as other transactions to be lost. The net result may be a degradation in performance. Therefore, a preemption decision in a real-time database system should be made carefully, and not necessarily based *only* on relative deadlines.

In our approach the preemption decision is made between two conflicting transactions; in the most general case, we propose that the decision be a function of deadlines, criticalness, elapsed times, total amount of computation time consumed, and laxity approximations (if known) of the two competing transactions. We hypothesize that for many environments the decision criteria for preemption will not need to be a function of all these parameters, but may simply take deadline and criticalness into account.

We now propose new access control protocols which directly account for the time constraints of transactions, and performs preemption based on the criteria listed above. Hence, the new protocols should have significantly better performance than

[2]In this paper we consider that preemption is the equivalent of aborting the transaction. While, for some cases, by using semantic information and/or forward error recovery techniques it might be possible to avoid completely aborting a preempted transaction, in this paper we do not consider these issues.

a traditional protocol in a real-time database system[3].

3.2 The Virtual Clock Access Protocol

The following is a brief outline of our new access control protocol based on the notion of a virtual clock [4,9]. We discuss the implications of the protocol in the next subsection.

The Virtual Clock Access Control Protocol

- Assume that according to transaction criticalness, transactions are categorized into n classes. The smaller the class number of a transaction, the less critical it is.
- Each transaction, T_i, has a virtual clock associated with it. The protocol controls the setting and running of the virtual clocks. VT(T_i) denotes the virtual clock value for transaction T_i. When transaction T_i starts, VT(T_i) is set to the current real time, say t_{s_i} — the start time of transaction T_i. Then, the virtual clock starts to run at rate β_k where k is transaction T_i's class number.
$$\beta_1 \leq \beta_2 \leq \beta_3 \ldots \leq \beta_n.$$
That is, the more critical a transaction is, the faster its virtual clock runs. Let t be the current time, then at any time $t \geq t_{s_i}$,
$$VT(T_i) = t_{s_i} + \beta_k(t - t_{s_i}).$$
- If transaction T_i requests access to data item, D, and this data item is not currently being used by any other transaction, then transaction T_i is allowed to access the data.
- Assume that at time t, while transaction T_i is accessing data item, D, another transaction, T_j, requests access to D. If the T_j's deadline is larger than T_i, T_j waits. Otherwise, if VT(T_i) is less than T_i's deadline, T_j preempts T_i.
- When T_i completes its access to a data item, if more than one transaction is waiting for access, the one with the minimum deadline accesses the data first.

[3]We have performed extensive simulations on real-time task scheduling. Those results clearly show the superiority of any scheduling algorithm that explicitly accounts for deadlines over algorithms that ignore task deadlines. Similar results can be expected for real-time access control, although, to date, we have not performed such simulations.

3.3 Remarks

In this virtual clock access protocol, one transaction may preempt another based on their relative deadlines, and on the criticalness and elapsed time of the running transaction. Also note that when the virtual clock of an executing transaction has surpassed its deadline, it cannot be preempted. Intuitively, this means that for this running transaction to make its real deadline, we are predicting that it cannot be preempted. Before discussing a generalization of this version of the protocol, let's consider a few special cases to help understand the power of the protocol: Assume T_i is running, and T_j requests a data item used by T_i in a conflicting mode. Also assume that both T_i and T_j are in the same class, say class k.

1. If $\beta_k = 0$, then $\mathrm{VT}(T_i) = t_{s_i}$ which is the start time of T_i. Because t_{s_i} is always less than T_i's deadline, T_i is preempted if T_j's deadline is less than T_i's even if T_i has been lost. That is, with $\beta_k = 0$, the protocol implements a *minimum-deadline-first* policy for transactions in class k regardless of whether T_i has already surpassed its deadline.
2. If $\beta_k = 1$, then $\mathrm{VT}(T_i) = t$ which is the current real time. T_j can preempt T_i if T_j's deadline is less than T_i's and T_i has not missed its deadline. The protocol now implements *minimum-deadline-first* policy before the deadline is surpassed, but does not allow preemption after the deadline.
3. If $\beta_k = \infty$, then $\mathrm{VT}(T_i) = \infty$. Hence, T_i is never preempted no matter what T_j is. That is, the protocol presents a *non-preemptive policy.*

We believe that setting all $\beta's$ a priori to one of the above three cases will not give good performance. Our protocol provides the opportunity to adjust the values of the β's.

In general, our protocol synthesizes a number of considerations.

1. Information on the amount of elapsed time for transactions $(t - t_s)$ is used in the preemption decision. That is, if a transaction has existed for only a short time it is more likely to be preempted, and vice versa. Note that the amount of computation time used by the transactions are implicitly taken into account here. The larger computation time used, the larger the elapsed time may be.
2. The relative deadlines of transactions are used. Only if the running transaction has a larger deadline is preemption considered. Further, the larger the deadline of the running transaction is the more likely it will be preempted.

3. The criticalness of the running transaction is also used in making the decision. However, the above protocol does not consider the criticalness of the requesting transaction, T_j. This is a weakness of this protocol and will be overcome by an alternative protocol described in the next two sections.

The proposed protocol can easily be extended. For example, one may want to explicitly account for the amount of computation time used by the transactions, because all else being (almost) equal, we do not want to preempt transactions that have received a significant amount of service. Further, one may also like to take into account the laxity information. It is usually very difficult to determine the computation time and laxity of a transaction. However, if an estimate of laxity were available, it might prove useful in making some decisions. For example, if a transaction has a very long laxity, then it it is likely that it can be preempted, restarted later, and still make its deadline. Also, it is necessary to account for the overhead of aborting and possibly restarting.

Hence, to be more effective, the above protocol may be generalized to take into account all kinds of information such as the deadlines, criticalness, elapse times, computation times required, computation times used, laxities, abortion/preemption overheads, etc. of the involved transactions. We argue that the algorithm used for making the decision of preemption must be very fast because it is used on-line.

In addition to the above protocol, we also considered attaching a virtual clock to the data items instead of the transaction. Such an approach has several anomalies that make it seem unattractive. Due to lack of space we do not describe this alternative protocol here.

3.4 An Access Protocol Based On Value Functions

The Pairwise Value Function Access Control Protocol

- Assume that according to transaction criticalness, transactions are categorized into n classes. The smaller the class number of a transaction, the less critical it is.

- Each transaction, T_i, has a value function associated with it. When two transactions conflict over a data item, the protocol computes the value functions of competing transactions and compares them. Let $VU(T_i)$ denote the value function for transaction T_i. Let γ_k denote the criticalness of a transaction where k is transaction T_i's class number.

$$\gamma_1 \leq \gamma_2 \leq \gamma_3 \cdots \leq \gamma_n.$$

Let t be the current time, then at any time $t \geq t_{s_i}$,

$$VU(T_i) = \gamma_k(w_1(t - t_{s_i}) - w_2 d + w_3 c - w_4 l).$$

```
where
  d    = deadline
  c    = computation time consumed by the transaction
  l    = laxity approximation, if known

and the w's are weighting factors.
```

- Assume that at real time t, a real-time transaction, T_i, requests access to data item, D. If this data item D is not currently being used by any other transaction in a conflict mode, then transaction T_i is allowed to access the data.
- While transaction T_i is accessing data item, D, another transaction, T_j, at real time t' where ($t' > t$), may request access to D. If the value of the function $VU(T_j)$ is less than or equal to the value of the function $VU(T_i)$, T_j waits, else if greater, T_j preempts T_i.
- When T_i completes, if more than one transaction is waiting for access, the one with the maximum $VU(T_j)$ accesses the data first.

3.5 Remarks

In this access protocol, one transaction preempts another based on the above general formula. By appropriately setting parameters or weights to zero it is easy to create various the situations, e.g., where a smaller deadline transaction always preempts a larger deadline transaction. This value function approach synthesizes a number of considerations and is efficient since it needs to compare only two values at any one time. First, information on the amount of elapsed time for transactions $(t - t_{s_i})$ is used in the preemption decision. That is, if a transaction has existed for only a short time it is more likely to be preempted, and vice versa. Two, the relative deadlines of transactions are used. If a transaction has a very large deadline, then the formula may permit its preemption. Three, the relative criticalness of the competing transactions is also used in making the decision. This is a major distinction between the two versions of the algorithm. We anticipate that criticalness and deadlines are the most important factors. Four, the amount of computation time used by the transactions is also important because, all else being (almost) equal, we do not want to preempt transactions that have received a significant amount of service. Five, in general, it is usually very difficult to determine the computation time and laxity of a transaction. However, if an estimate of laxity were available, it might prove useful in making some decisions. For example, if a transaction has a very long laxity, then it it is likely that it can be preempted and restarted later and still make its deadline. Six, it is necessary to account for the overhead of aborting and possibly restarting. While this is difficult to do very precisely, we could attempt to do this by adapting the parameter γ. That is, the value of γ itself can change during the execution of the transaction. For example, using semantic information about a transaction, that transaction might be considered non-preemptable by appropriately setting the value of γ. Also, setting γ to a low value after a transaction's deadline has passed could account for the diminished value of the transaction at that time.

In summary, the two access control protocols differ in the following way. The virtual clock protocol is simple, has a small number of parameters, but does not take into account the criticalness of the requesting transaction. The value function protocol has more parameters but does consider all the necessary information. Both protocols have small overhead. In the future, we intend to study the performance of each protocol and identify workloads and environments where each works well.

4 Primitives for Hard Real–Time Data

Much real–time data is lower level than typically found in database systems. Such data often does not require the same synchronization, atomicity, and permanence properties of transactions. Our approach is to categorize real–time data into classes depending on their time, synchronization, serializability, atomicity, and permanence properties. Then, using the semantics of the data and the applications, we are attempting to developing special real–time transactions that supply only the minimal properties necessary for that class. This will reduce the system overhead in supporting access to various types of data and help verify that timing constraints can be met. In the remainder of this section we provide a detailed example of a hard real–time primitive called real–time stream, and discuss several other types of real–time data.

4.1 A Real-Time Stream

Atomic transactions can be considered programs with embedded concurrency control and failure recovery protocols. The structure of atomic transactions has been shown to be a very useful one for distributed database systems, because the embedded protocols allow application programmers to concentrate on the logic of computation rather than the problems of concurrency control and failure recovery. There is significant system overhead associated with supplying such a powerful primitive, the transaction, to database programmers. Real-time data does not typically require full concurrency control and failure recovery support and cannot afford the overhead costs. For example, a common type of real-time data is data being obtained from a sensor in a continuous stream, and the data being filtered through a number of stages (transformations). At one or more points the data from multiple streams are correlated. The correlated data is then used to make decisions. Consistent views of this information might then be necessary. Further, the important properties of such real-time data include a time window $< a, b >$ at each stage where a indicates the time by which data is to be available, and b the time by which it must be processed; an ability to correlate the data from multiple streams with respect to some time reference (timestamp or index); and upon failures the ability to resynchronize the streams, flush the streams, ignore a portion of the streams, or delay processing. To support the minimal set of properties needed at each stage we developed the notion of a real-time stream. This approach allows us to attain fast processing and keeps overhead as low as possible while retaining the well structured design concept of transactions.

More specifically, a real-time stream is composed of 3 sets of layers: the filter layers, the correlation layers, and the transaction layers.

- Filter Layers: There can be multiple filter layers each with a real-time constraint $< a, b >$. Typically, the filter layers have very tight deadlines and these layers abide by our segmentation rules [7] by preallocating processors and time to these filter layers. They would not be part of any dynamic guarantee routine, and they do not need the overhead of the concurrency control and failure recovery.

- Correlation Layers: For each correlation layer, multiple agents exchange data, correlate it, and draw some conclusions, thereby producing data for the transaction layer. Time constraints exist at the correlation layers, but are not as tight as at the filter layers. Depending on the application and system environment there might be a preallocation of resources or a dynamic guarantee to achieve the time constraints. However, there is usually no need for atomic actions at the correlation layer. The correlation process itself is working with approximations and missing data, so there is no need to expend system overhead to insure absolute consistency. What is required is a synchronization protocol to ensure that multiple data streams coming from different sources are synchronized. This protocol must be robust in the sense that data streams will be automatically re-synchronized even if the synchronization were disrupted by errors.

- Transaction Layers: Some of the data (or decisions) produced by filtering and correlation might need the full support of transactions. For example, all sites might have to consistently agree where a target is, what the target is, and what actions to initiate. In this case it is necessary to expend the system overhead to insure consistency and failure recovery. However, at this stage we would like to make the overhead as little as possible and maximize concurrency. To accomplish this we suggest tailoring transaction support to the problem at hand, resulting in a number of *lightweight* transactions, e.g., transactions which only support concurrency control, or just recovery, etc. The timing constraints at the transaction layer are typically more lax than at the other layers, and can be treated a soft real-time constraints, not subject to guarantees.

Real-time stream data as described above is one example of real-time data. It is difficult to define real-time data because sometimes it is the property of the data itself, and sometimes the property of the task that uses the data. Our informal definition of real-time data is any data that must be manipulated in some way (read, written, inserted, deleted, etc.) subject to a timing constraint. Therefore, stream data is real-time data because at each stage in the stream the data must appear and depart subject to timing constraints. Note that it is possible to have stream data where there are no explicit timing constraints. Other typical types of real-time data found in real-time systems include 1) write only data - here data is written to a log or output device at a certain rate or based on other timing constraints, 2) read only data - here data is read from a database or device with timing constraints, 3) write at the end of a file, but read in earlier parts of the file only, again based on timing constraints, and 4) one stage streams. Fortuneately, these types of real-time data usually have minimal concurrency problems (unique updaters) and various techniques can be used to relax timing constraints and minimize or even eliminate conflict.

5 Summary

In this short paper we discuss the notions of soft real–time transactions and hard real–time data. We propose two access protocols for soft real–time transactions and a stream primitive for hard real–time data. We also discuss the use of semantic information in real–time computing, and give some examples.

References

[1] Biyabani, S., "Criticalness Considerations in Task Allocation in Hard Real–Time Systems," Masters Thesis, Univ. of Mass, in preparation.

[2] Blazewicz, J., "Deadline Scheduling of Tasks with Ready Times and Resource Constraints", *Information Processing Letters*, Vol. 8, No. 2, February 1979.

[3] Leinbaugh, D., "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Trans on Soft Eng*, Vol. SE-6, January 1980.

[4] M.L. Molle and Lenonard Kleinrock, "Virtual Time CSMA: Why Two Clocks are Better than One", *IEEE transactions on Communications*, Vol. COM-33, No. 9, September 1985.

[5] Ramamritham, K: and J. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, Vol. 1, No. 3, July 1984.

[6] Stankovic, J. and K. Ramamritham, "The Design of the Spring Kernel," *Proc 1987 Real Time Systems Symposium*, 1987.

[7] Stankovic, J., and L. Sha, "The Principle of Segmentation," Technical Report, 1987.

[8] Zhao, W., Ramamritham, K., and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, May 1987.

[9] W. Zhao, and K. Ramamritham "Virtual Time CSMA Protocols for Hard Real–Time Communication", *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 8, August 1987.