On Building Distributed Soft Real-Time Systems

Ben Kao Hector Garcia-Molina Brad Adelberg

Department of Computer Science
Stanford University
Stanford, CA 94305
{kao,hector,adelberg}@cs.stanford.edu

Abstract

When building a distributed real-time system, one can either build the whole system from scratch, or from pre-existing standard components. Although the former allows better scheduling design, it may not economical in terms of the cost and time of development. This paper studies the performance of distributed soft real-time systems that use standard components with various scheduling algorithms and suggests ways to improve them.

Keywords*: soft real-time, distributed systems, deadline assignment, priority assignment, scheduling.*

1 Introduction

Consider a distributed system for stock market analysis and program trading. In this application, information on stock prices may be gathered from multiple information sources and piped through a series of filters for refinement. The refined information may then be stored in a database server to be queried by, for example, an expert system that spots trading opportunities. The expert system may then trigger certain buy-and-sell actions to realize a profit.

There are two interesting features to observe. First, the system involves many components of different types: multiple information sources supplying streams of financial data, a network component, a database server (probably I/O bound), and an expert system (probably CPU bound). Second, there are global tasks which consist of multiple stages involving work at multiple components. These tasks may have soft deadlines, e.g., a buy-sell action triggered by a stock price update should be implemented within 2 minutes from the time when the update arrives.

When designing a distributed real-time system, such as the program trading one above, one approach is to build it from scratch, coding each component with algorithms and features that match the applications' needs. However, the development and maintenance cost of such a system will be extremely high, and the development cycle will be very long. Alternatively, one can build the system by piecing together well-tested, easy-to-maintain standard components, such as a commercial database server, a token ring network, etc. The disadvantage of this approach is that standard components are usually not designed to handle real-time tasks. As an example, commercial database systems do not compare the deadlines of conflicting transactions when granting a lock request, which may result in priority inversions. Even if standard real-time components are available (e.g., commercial real-time operating systems), they may vary in their ability to meet tasks' real-time requirements. Also, their schedulers may not provide any external control. This makes global real-time scheduling algorithms impossible because they demand cooperation and coordination among the local schedulers. Given these difficulties, is it possible to construct a distributed system out of a variety of conventional non-real-time and real-time components and still meet real-time constraints? Even if we cannot meet all real-time constraints, can we ensure that "almost all" are met? In this paper we address these questions. Although we will not provide definitive answers, we do present the results of some simple experiments that provide insight into those questions. In particular, the results illustrate the cost (in terms of missed deadlines) of using conventional, heterogeneous components, and suggest strategies for reducing the cost.

For our experiments we consider a distributed system with a number of components. Each component employs its own scheduling policy (whether real-time or non-real-time). We study how the local schedulers impact the system's ability to meet task deadlines. We also suggest ways of improving the system performance when some of the components cannot perform real-time scheduling. Before we proceed, we state some premises:

• We focus on *soft real-time* systems. In such systems, a primary performance goal is to meet as many deadlines as possible, but unlike hard real-time systems, there is no absolute guarantee that all deadlines will be met. There are two reasons why we look at soft (instead of hard) real-time systems. First, the kinds of distributed tasks we are looking at are quite complex, involving multiple stages of processing. This means that it is generally hard to get the accurate running time estimates that are required for hard real-time scheduling. Second, in many applications it is undesirable or impossible to place an upper bound on the load. Both problems make hard real-time scheduling impossible to achieve.

• Each component's scheduler is independent. There is no global scheduler that instructs each scheduler what to do. Each scheduler makes decisions based solely on the subtasks that have been presented to it for execution, without consulting other schedulers. We believe that large systems are built out of preexisting components. Each component will have its own scheduling policy and will be unable or unwilling to coordinate or subordinate its scheduling decisions with (or to) others.

• We look at *on-line* scheduling, as opposed to a priori scheduling when the tasks are defined or first submitted. On-line scheduling is more appropriate for the type of systems we study where the tasks' types or their durations may be unknown in advance. Also, when the system provides distribution transparency (e.g., whether a data item is available locally or remotely), the subtasks to be created are unknown until run-time and thus off-line pre-analysis is not appropriate.

• Each system component is unique. If a task is scheduled to run at a particular component, it must run there. There is no load balancing, i.e., an overloaded component cannot ship tasks to other components.

The rest of this paper is organized as follows. Section 2 describes the model for our study. Section 3 discusses the case when all system components use real-time scheduling such as earliest-deadline-first. In Section 4 we study how the system performance degrades when some of the components serve tasks in FCFS order. In Section 5 we discuss the benefits we can gain when a non-real-time component provides static priority scheduling. Finally, we conclude our paper is Section 6.

2 The Model

In this section, we describe the task model, the system model, and the simulation model we use for our analysis. We will first define global tasks, and then describe a model of a distributed system on which tasks are mapped for execution. As the reader will notice, our model is quite simple. As mentioned earlier, our goal is to understand the basic tradeoffs, not to realistically evaluate a particular system.

2.1 The Task Model

In this paper, we consider *serial* global tasks that involve work at multiple components in the system. As shorthand, we use the notation $T = [T_1 T_2 ... T_n]$ to represent a *global* task T that consists of n subtasks, T_1, T_2, ..., T_n, to be executed in *series*. A subtask T_i ($i > 1$) cannot execute before subtask T_{i-1} finishes.

A task X (whether it is a subtask or a global task) has the following attributes:

$$\begin{aligned} ar(X) &= \text{arrival (or submission) time of } X, \\ dl(X) &= \text{deadline of } X, \\ sl(X) &= \text{slack of } X. \end{aligned}$$

We also define the *stage* of a subtask X to be its position in a global task. For example, if $T = [T_1 T_2 T_3]$, then $stage(T_2) = 2$. We say that a subtask X is an *earlier-stage* than another subtask Y if they are of the same global task and $stage(X) < stage(Y)$.

Finally, there is the issue of tardy tasks, or *overload management policy*. Suppose a task X has already missed its deadline, but has not completed execution. One option is to abort X as soon as it misses its deadline, under the assumption that whatever it was doing is now useless. A second option is to continue to process X, under the assumption "better late than never." Due to space limitations, in this paper, we only focus on the no abortion case.

2.2 The System Model

Our model of a distributed real-time system consists of a number of *components* (Figure 1). These components manage different resources like a database, an expert system, or a compute engine. Even the communication network is considered a resource and is subsumed as one or more components.

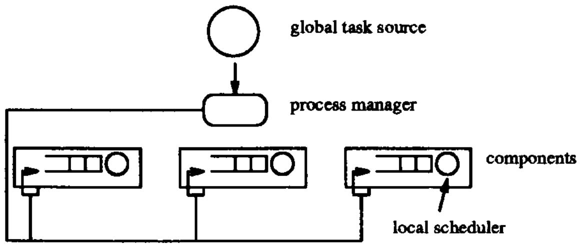

Figure 1: The System Model.

For example, a direct link between two sites is considered as one resource, while a LAN is considered another. Task service order is scheduled by a local scheduler residing at each component. These schedulers are all independent and do not collaborate. The only things that may influence scheduling decisions are the real-time attributes associated with each task.

Newly created global tasks are first processed by the process manager. Figure 1 shows a single process manager, but in reality there can be many. Each manager controls one or more global tasks. We assume that certain control information of a global task, such as the precedence relationship among the subtasks, and the end-to-end deadline, is available to its process manager. The major functions of the process manager are to submit the subtasks to the appropriate components for execution and enforce the precedence constraints among the subtasks of a global task. In some cases, the process manager has to generate scheduling information for a subtask before submitting it. For example, if a component A accepts tasks with priority (and schedules them according to the priorities) and another component B schedules tasks according to their deadlines, the process manager has to make sure that the subtasks submitted to the components have the right attributes and the *appropriate* values for these attributes. Part of our study concerns how these values are assigned and what the performance implication of a good assignment strategy is. In reality, the process manager consumes system resources (e.g., communication overhead between the manager and the components,) but this consumption can be modeled as additional subtasks, and be handled similarly. Therefore, we do not model the resource requirement of the process manager explicitly.

2.3 The Simulation Model

In order to study and contrast system behavior under different local scheduler algorithms, we developed a simulation model and performed extensive experiments. Our simulator is written in the simulation language **DeNet**[6]. Each simulation experiment (generating one data point) consists of two simulation runs, each lasting one million time units (at least 100,000 tasks are generated per run, many more for high load experiments). The 95% confidence interval is $\pm$ 0.35 percentage point (much smaller for high load experiments) for the missed deadlines figures shown in later sections.

The structure of our simulation model follows the conceptual model described in Sections 2.1 and 2.2 with the following characteristics.

Components: There are k components in the system. Each component schedules its tasks using a local scheduler. In this paper, we consider three types of local schedulers: earliest-deadline-first, static priority, and FCFS. We assume a *no-preemption* scheduler in all cases.

Global Tasks: Global tasks are generated as a *single stream* of a Poisson process with mean inter-arrival time $1/\lambda_{global}$. In order to simplify our discussion, we hold the simple view that global tasks are homogeneous. In particular, we assume that all global tasks consist of m subtasks and the execution times of the subtasks all follow the same exponential distribution with mean equal to $1/\mu_{subtask}$ time units. The total execution times of global tasks thus follow an m-stage Erlang distribution with mean $m/\mu_{subtask}$. The rate of work due to global tasks is therefore $m\lambda_{global}/\mu_{subtask}$. We assume that nodes are equally likely to be chosen as the execution node of a subtask. Slack of global tasks is uniformly distributed in the range $[S_{min}, S_{max}]$

System Load: We define the *normalized load* (or *load* for short) to be the ratio of the rate of work generated to the total processing capacity of the system. That is,

$$load = \frac{m \cdot \lambda_{global}}{k \cdot \mu_{subtask}}.$$

For a stable system, we have $0 \leq load < 1$. Table 1 shows the parameter setting of our baseline experiment.

3 Real-Time Schedulers

For the first scenario, we assume that all components are capable of doing earliest-deadline-first (EDF) scheduling. Before a subtask is submitted to a component for execution, the process manager must assign a deadline to the subtask so that the component knows how to schedule it. One very simple way of assigning the subtask deadline would be

Overload Management Policy	No Abortion
$\mu_{subtask}$	1.0
k (# of components)	6
m (# of subtasks of a global task)	4
load	0.5
$[S_{min}, S_{max}]$	[5,20]

Table 1: Baseline setting

to let the subtask inherit the (end-to-end) deadline of its global task. We call this the *Ultimate Deadline* strategy (UD). Previous studies [5, 7] show that UD works fine if all the global tasks in the system are of similar length (in terms of number of subtasks and execution time). Otherwise, long global tasks will suffer a very high missed-deadline rate. The reason why UD performs badly in the presence of short tasks is that a long global task, with its many stages of subtasks, usually has a later deadline than a short one does. When an early stage subtask (T_i) adopts its global task's late deadline, the local EDF scheduler will be tricked into believing that T_i has a lot of slack to burn. Even the time that should be reserved for the execution of the later stage subtasks is considered slack to T_i. The EDF scheduler will thus give a very low priority to T_i. This results in an exceptionally long delay in T_i's execution and may cause the long global task to miss its deadline.

The deadline assignment problem is studied in various contexts in [5, 7, 4]. In particular, in [5], several simple strategies for reducing the missed-deadline rate of global tasks are discussed. One strategy called *Equal Slack* (EQS) estimates the amount of slack that a global task has and divides it among the remaining subtasks equally. For example, consider a global task that arrives at time 0 with a deadline of time 12 and that has four subtasks, each of which is expected to run for 1 unit of time. This global task has, in total, 8 units of slack. EQS would allocate 2 units of it to the first subtask (reserving the other 6 units to later stages), and assign a sub-deadline of time 3 to the first subtask. It has been shown in [5] that EQS results in a much lower global task missed rate. In this paper, we assume that whenever a subtask is submitted to a component that does EDF scheduling, it is assigned a sub-deadline according to the EQS strategy.

4 FCFS Schedulers

To build a soft real-time system, one may or may not be able to use specialized real-time components, such as a real-time network or a real-time database system that schedules I/O requests according to deadlines. Real-time components would obviously make it easier to meet system deadlines. Unfortunately, they may be much more expensive and may not be readily available. For instance, standard networks such as Ethernet or token ring, and their corresponding drivers do not support real-time traffic, and they tend to deliver messages in an FCFS order. As another example, *sophisticated* disk controllers that employ algorithms such as the Elevator Algorithm [8] schedule disk requests according to disk block position instead of request deadlines. Conventional real-time scheduling algorithms like earliest-deadline-first do not perform well for disk scheduling because they cause long average seek time and poor throughput [3, 1]. If one cannot afford the use of real-time components, how will that hamper the efficacy and performance of the real-time system? Is it crucial that every single component in the system understands deadlines? Will a non-real-time component cause a "deadline bottleneck"? If an urgent task experiences a long delay at a non-real-time component, can it catch up while visiting real-time components? In this section we look at how the performance of a distributed soft real-time system degrades when some of the components perform FCFS scheduling instead of EDF.

We conducted an experiment according to the model and baseline setting in Section 2 and measured the *fraction of missed deadlines* (MD), or miss rate for short, of global tasks as the system load changes. In the experiment, we randomly pick N_{fcfs} components to use FCFS local schedulers, while other components use EDF. Figure 2 shows the result of the experiment. Three curves are shown corresponding to $N_{fcfs} = 0$ (all local schedulers are EDF), 1, and 6 (all local schedulers are FCFS).

From the figure we see that global task miss rate increases as the system load increases, as expected. At low load, the system has enough capacity to handle tasks so whether the local schedulers use FCFS or EDF is inconsequential. However, as the load increases and the system starts to miss more deadlines, the difference between the two schedulers becomes more significant. At high load (*load* $= 0.65$), a system with 6 FCFS local schedulers misses 22.4% of the deadlines — four tenths more than a system with 6 EDF schedulers, which misses about 16.1% of

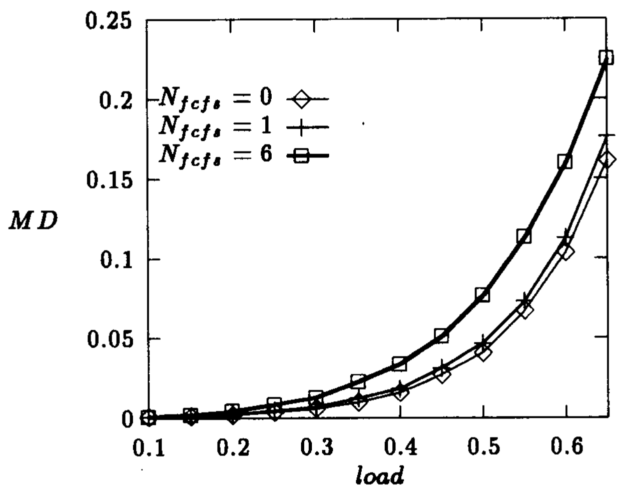

Figure 2: Fraction of missed deadlines versus system load for various values of N_{fcfs}

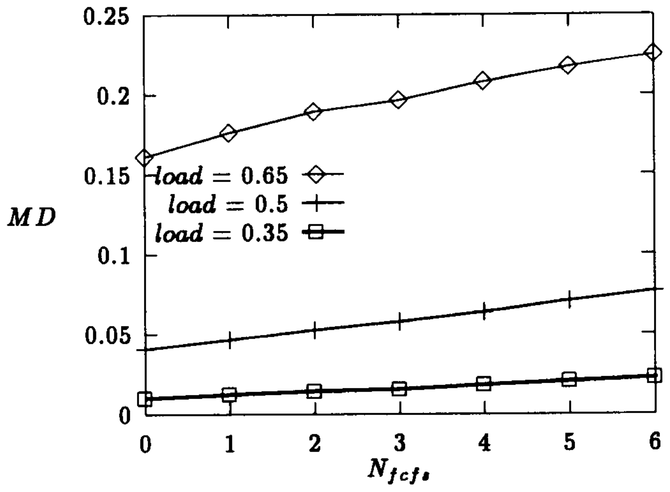

Figure 3: Fraction of missed deadlines versus number of FCFS local schedulers

the deadlines. We would like to remark that under normal condition, a soft real-time system should be operating with low load and few deadlines missed. However, occasionally the system will be overloaded, and it is precisely at those times when we need a good scheduler that misses the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Figure 2 are important.

While it is not surprising to see that an all-FCFS system performs poorly compared with an all-EDF system, it is interesting to see that switching *one* component from EDF to FCFS only causes a mild performance loss. In fact, in the baseline model, the performance of the system appears to degrade linearly with respect to the number of FCFS components. This is shown in Figure 3 which plots MD against N_{fcfs} for various values of system load. The phenomenon that performance degrades gradually in an almost linear fashion is somewhat counterintuitive. For instance, one would expect that with a single FCFS server, the other 5 could "make up" with their good scheduling. For example, if a task is delayed at the FCFS server, perhaps its could "make up for the lost time" when it visited the remaining EDF components. If this argument held, then the degradation for adding one FCFS server would be less than 1/6th of the total degradation when all 6 servers are FCFS. Unfortunately, Figure 3 shows that this argument does not hold.

The almost linear degradation of performance is due to the fact that all of the components in the baseline model handle a similar amount of load and that they work for a similar number of global tasks. To see how the system behaves under different loading condition, we modify our baseline experiment so that all the stage 1 subtasks go to component 1 exclusively while components 2 through 6 handle the rest of the subtasks. (For example, every global task in a distributed soft real-time system needs to use the network component but not all of them query the database server.) Since each global task consists of four subtasks (Table 1), the load is biased with component 1 handling 25% of the system load and the other 5 components handling 75% of the load (or 15% each). In this experiment, we switch the local schedulers from EDF to FCFS one by one. In one case, we start the switching from component 1; In another case, component 1 is the last one to be switched. Figure 4 shows the results of the experiment.

In Figure 4 two pairs of curves are shown ($\{+, \times\}$ and $\{\diamond, \square\}$) corresponding to two levels of system load (*load* = 0.5 and *load* = 0.35). For each load, we show the MD for two cases — switching component 1 first or last. Let us focus our attention on the top most curve (line "+"). We will not discuss the other three curves since similar conclusions can be drawn from them. From the "+" line we see that switching the more heavily loaded component (component

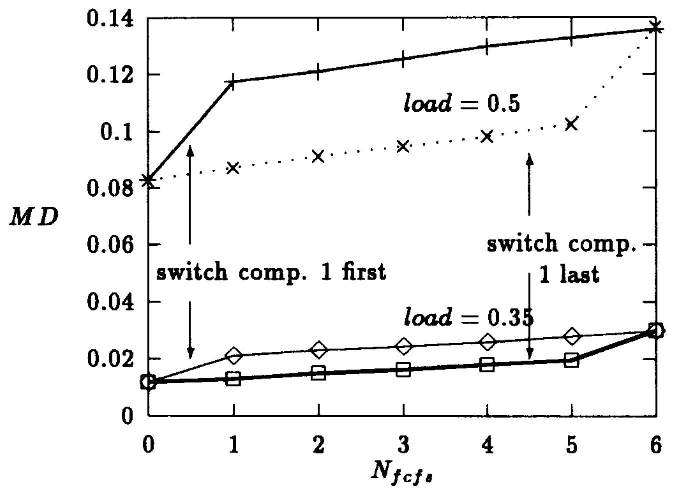

Figure 4: Fraction of missed deadlines versus number of FCFS local schedulers

1) from EDF to FCFS causes the miss rate to jump from 8.2% to about 11.7% (or by 3.5%) while switching the other five causes a gradual linear increase in MD to 13.5% (or by 0.36% stepwise). Even though components 2 to 6 together are responsible for 75% of the total system load, Figure 4 shows that it is actually *less* beneficial to run five EDF schedulers at these components than it is to run one EDF scheduler at component 1.

We have tried other system parameter settings, such as changing the slack range, changing the number of components in the system, directing all the *stage 4* subtasks to component 1 instead of stage 1 subtasks, etc. In all these cases we obtain similar results. This lets us conclude that the performance loss of having a FCFS component instead of an EDF one is directly related to the load of that component. When all the components in the system are loaded similarly and are used by relatively the same number of tasks, this performance loss is almost linear with respect to the the number of FCFS components. In this case, it is not absolutely critical to have every single component in the system use deadline scheduling. It may be more economical to use a couple of FCFS components if a mild performance loss is tolerable. In case the loading is biased, the designer should concentrate his effort in the more heavily loaded components. To ensure better performance, these components should use real-time schedulers or should be duplicated to increase their capacity. In some cases, the performance gain obtained by converting a heavily used component to EDF scheduling can be more significant than converting many other less heavily used ones.

5 Static Priority Schedulers

In the previous section we discussed how system performance degrades when real-time components are replaced by FCFS components. Very often, standard non-real-time components can do better than FCFS by scheduling tasks using static priority (e.g., POSIX UNIX and token ring networks). In this section we discuss how we can utilize these static priority schedulers in a distributed soft real-time system, and how much performance improvement one can achieve over FCFS schedulers.

One idea is to emulate a real-time scheduling algorithm using static priority. In [2], different ways of mapping the real-time attributes of a task, such as its deadline, arrival time, slack, etc., into a priority level on a ***uniprocessor system*** are studied. Here we mention a simple one called ***Earliest Deadline Relative*** that can be easily adaptable to our distributed system. Before a subtask T_i is submitted to a component C_j, the process manager first computes T_i's sub-deadline $dl(T_i)$ using EQS (see Section 3). The sub-deadline is then used to compute a priority level for T_i according to a linear mapping:

$$priority(T_i) = (dl(T_i) - ar(T_i))/ts_j$$

where ts_j is a tuning parameter for component C_j. Due to space limitation, readers are referred to [2] for how to determine good ts values.

We implemented the mapping function in our process manager and studied performance. Figure 5 compares the global task miss rate under three scenarios: (1) all-FCFS, (2) all-EDF, and (3) all components use static priority scheduling with 4 priority levels, and the process manager uses the real-time emulation mapping with the ts's set to 4.5 (RTE).

From the figure, we see that even with only four priority levels, static priority scheduling with RTE performs comparably to EDF. At higher load, RTE actually outperforms EDF slightly. The reason why RTE misses slightly fewer deadlines is that first of all, RTE compares tasks according to how much time they have $(dl(T_i) - ar(T_i))$ instead of their absolute deadlines. Second, under high load, EDF tends to give preference to tasks that are late. On the other hand, because of its coarse priority granularity, in overload situations RTE cannot order all tasks properly by deadline. Thus, RTE essentially gives non-

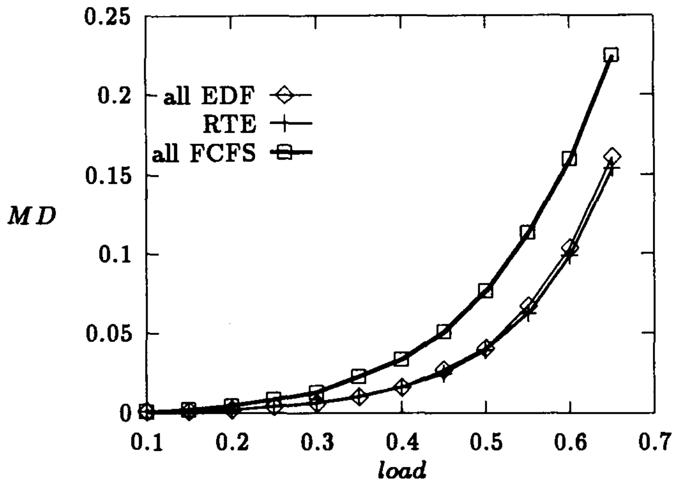

Figure 5: Performance of RTE

tardy tasks a chance to finish before their deadlines. Further analysis on the performance of RTE in a uniprocessor environment can be found in [2].

The disadvantage of the RTE approach is that its performance is sensitive to the *ts* value and the distribution of the tasks' slack. In our baseline model, the slack distribution is uniform and thus a linear mapping works well. For other slack distributions, different mappings that can evenly distribute the task population into the limited priority levels are needed to ensure good performance. This adds to the complication of the process manager. However, when a real-time component is not available, RTE with even a simple mapping is an effective and economical way of reducing missed deadlines.

6 Conclusion

In this paper we have explored a non-traditional type of real-time system where one has to use some conventional non-real-time components due to cost or availability considerations. Of course, in a *hard* real-time system, one cannot use any non-real-time components. However, we believe there are many applications where deadlines are soft and one does wish to consider the various cost vs availability of components vs real-time performance tradeoffs. Through simulation studies, we have illustrated how system performance might degrade when EDF schedulers are replaced by FCFS schedulers. In general, when the system components are evenly utilized and their loads kept low, using a few non-real-time components does not overly increase the miss rate. It may be more cost effective to use a few standard components when specialized real-time ones are expensive. However, for systems that use one component more heavily than the others, it is important that the highly loaded components be equipped with real-time schedulers. In case real-time schedulers are not available, we should either increase the capacity of these components (for example, by duplication) to minimize the effect of FCFS scheduling, or in case static priority schedulers are available, use a real-time emulation algorithm that maps deadlines to priorities. Our results suggest that the latter method will perform very similarly to a real EDF scheduler.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 113–124, 1990.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *IEEE Real-Time System Symposium*, 1994.

[3] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of the 15th VLDB Conference*, pages 397–410, 1989.

[4] B. Purimetla et. al. Priority assignment in real-time active databases. In *Proceedings of IEEE Parallel and Distributed Information Systems*, pages 176–184, 1994.

[5] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437, 1993.

[6] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[7] H. Pang, M. Livny, and M. J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of IEEE Real-Time Systems Symposiuim*, 1992.

[8] J. L. Peterson and A. Silberschatz. *Operating System concepts*. Addison-Wesley, 1985.