Experimentation in CPU Control with Real-Time Java

Gerasimos Xydas
BT Advanced Communications Technology Centre
gxydas@di.uoa.gr

Jérôme Tassel
BT Advanced Communications Technology Centre
jtassel@jungle.bt.co.uk

Abstract

This paper describes experiences in using an O.O. language (Java) in designing, prototyping and evaluating a CPU manager. QoS Animator facilitates the execution of object oriented Java applications with time requirements and provides protection mechanisms to preserve system's integrity against untrusted code. It is adapted to the system performance and provides a rate-monotonic based scheduling algorithm, WCET calculation at run-time and protection from high-CPU-consuming and "bad" code. The introduction of the Low Frequency Filter enhances the timeliness offered to applications in general-purposes Operating Systems (OS). The evaluation was done with a Windows NT specific prototype and proved successful.

1. Introduction

Internet is naturally unsuitable to deliver data with time constraints. Guarantees in doing so are provided through QoS control in the network and the end-system for sharing resources among concurrent applications and streams. Some emerging IETF protocols deal with network QoS provision [1][2][3]. However, QoS must be provided in every element that is involved in the delivery of the data (CPU, buffers, i/o, etc) [4]. Applications that run on host machines are written without resource sharing in mind and the amount of resources they require is increasing as the available bandwidth increases. Moreover, mobile code might affect the system' s integrity (especially in an Active Networks context), so mobile code should be accommodated in safe languages [5][6]. A resource manager in a host machine should serve concurrent applications according to their needs while protecting the system from untrusted code.

The most crucial host resource is the CPU. Some work that deal with CPU sharing are based on specific Real-Time Operating Systems (RTOS) [7][8], and they introduce the notion of CPU reservations by forcing tasks to meet their deadlines. RTOS provide the time-related facilities to do so. Other work introduces some barriers that prevent tasks from using more than a specific amount of CPU [6]. In this work we deal with a portable solution that share the CPU among Java applications with time critical sections and helps them to meet their deadlines, while protecting the system from "bad-code".

Real-time applications are usually developed using native languages and targeted for well-defined execution environments. Furthermore, a RTOS is required to provide timeliness and to efficiently handle interrupts. The portability of real-time code is subject to the ability to adapt to the run-time execution environment. Java is a portable O.O. language that was introduced to facilitate networked applications but it does not provide facilities to accommodate real-time code, in terms of semantics and predictable behaviour [9]. Real-Time Java introduces time and memory semantics to support real-time development and provides run-time analysis of the byte-code that adapts a real-time Java application to the execution environment [10][11]. However, native support can provide enhancements to traditional Java and extend its capabilities. In our work we take some minimal native support from the underlying OS, by boosting the priority of the JVM process to "real-time".

We chose to build our prototype on top of Windows NT rather than a research RTOS, as it is a widely used general purpose OS. Although it is not deterministic and thus unsuitable to serve applications with time constraints [12], it allows almost the exclusive use of the CPU by processes that run at the REALTIME_PRIORITY-_CLASS. According to [12] this priority level must be used with extreme care as it may considerably affect the system' s integrity.

In this work not only do we take advantage of this feature, but we also enhance the timeliness of NT in this priority level. The rest of this document describes the architecture of the QoS Animator and the results we achieved through comparison with conventional techniques.

2. The QoS Animator

QoS Animator is a CPU manager that supports Java applications with soft real-time requirements. It provides a simple API that enables them to include time critical sections in their code. The API is as simple as the invocation of the register(Profile) method of the QoS Animator class,

where the Profile describes cross platform attributes like the *period* of a task. Figure 1 presents the architecture of the QoS Animator.

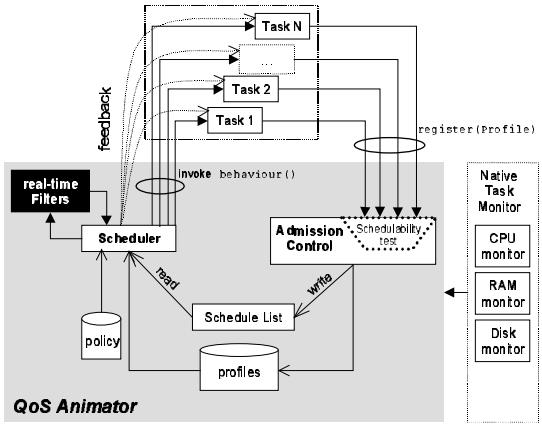

Figure 1

The prototype is currently running on top of Windows NT, but it does not require NT. The only support it gets is the boost of its process' priority to REALTIME-_PRIORITY_CLASS. The portability of the QoS Animator is subject to the availability of a feature that allows almost the exclusive use of the CPU by its process, like is being done with the REALTIME_PRIORITY_CLASS in NT.

2.1. Admission Control

The traditional *rate-monotonic algorithm* (RMA) [13] schedules tasks without taking their origin into account. It focuses on periodic tasks and does not accommodate interrupts and other asynchronous events that interrupt the normal execution of tasks. In order to deal with the lack of a RTOS and to tolerate interrupts, while keeping the simplicity of the RMA, we extend its *utilisation bound test* formula to:

$$\sum_{i=1}^{n} \frac{\lceil C_i \times E \rceil}{T_i} < \alpha \times n \times (2^{1/n} - 1)$$

The extended execution time that a task now has is:

$$C_i^{'} = \lceil C_i \times E \rceil$$

where C_i is calculated during runtime (see Runtime WCET calculation). E (namely `EXTENSUM`) is a metric that gives more time to tasks than actually needed and it is defined dynamically according to the workload by the CPU monitor [14]. This way a task deals with short interrupts, without missing any portion of the requested CPU time. It does not modify the *rate monotonic sort* as it is only used during the execution of a task to give it more time. Factor α limits the amount of the CPU allocated to real-time tasks, in order to preserve the normal execution of the OS. This formula actually shows that the CPU utilisation for real-time tasks will never be 100% (but 50-70%). We do not target on efficient CPU usage but on efficient usage of a portion of the CPU. By leaving time to the OS we ensure that time sharing applications and other OS activities will not starve and system integrity will be preserved. In [15] the RMA utilisation bound test formula has been extended by adding a factor U to indicate the timing unpredictability caused by the OS. This has the disadvantage that the factor U is extracted empirically and not dynamically, so it is not portable.

2.2. Scheduler with real-time filtering

The scheduler divides time in two categories: tasks' time and OS's time. The first one refers to real-time tasks, while the second refers to time-sharing applications and other OS activities.

The scheduling algorithm is very simple. It first seeks for the next task to be focused, based on the rate monotonic sort. The scheduler then sleeps for the duration that the tasks need to execute. If no task is scheduled (i.e. CPU is granted to the OS), the scheduler invokes the apply() method of all the filters that extends the RealTimeFilter class and adapts that pause time to the expected progress that the scheduler should have, by shortening the time of that pause. After that, the scheduler removes the focus from the current executed task (if any) and processes a feedback from it in order to ensure that its progress is consistent with its attributes. This feedback is also fed to the real-time filters.

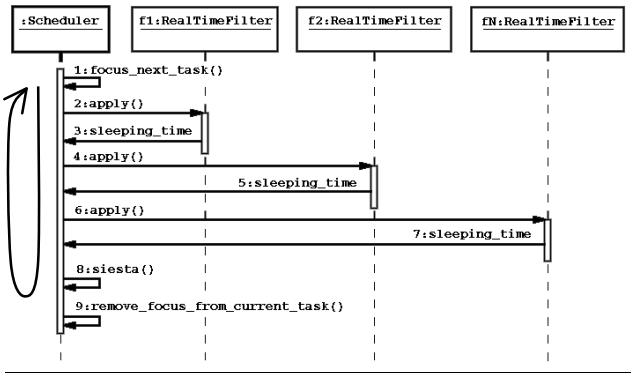

Figure 2

A general purpose OS does not provide the timeliness of a RTOS. However, we concluded that by monitoring the behaviour of the response times, we can enhance the timeliness offered to applications. Also, if a delay is experienced in one task, it propagates to other tasks in the same process. By capturing the delay caused in a task, we can prevent others from experiencing the same. Real-time filters are applied between the pauses of the scheduler to

make up for the missed real-time intervals and to prevent jitter and delay propagation. Normally, the amount of time that the scheduler sleeps is defined by:

- the time that is left until a higher priority task is ready to run,
- the WCET of the currently executed task,
- the time that is left until a lower priority task is ready to run.

However, delay is possible to happen, which can be detected by examining the feedback from the task. Feedback is generated when the scheduler removes the focus from it, no matter if the task has met its deadline or not. The Low Frequency Filter compares the achieved end time of task' execution with the expected one, as well as OS' s time intervals. If delay is experienced either in executed tasks (i.e. frames arrive at a lower frequency than they should) or in pauses between tasks (i.e. OS' s time), the filter adapts the scheduler' s sleeping time to the delay, giving smaller amounts of time to the next OS activity or real-time task. In the first case it just gets back the time that the OS grabbed from real-time tasks. In the latter, - as we have already taken care so that the execution time allocated to a task is bigger than the actually time needed (EXTENSUM) -, we have the flexibility to shorten this time, as such delays are short compared to the extra allocated times [12].

2.3. RM simulation

RMA can not be applied directly to Windows NT as NT offers only 7 level of thread priorities within a process, or 32 level of thread priorities in all process levels [12]. Note that a task in QoS Animator is assigned to a thread, but the design allows assigning it to a group of threads. As we target a single JVM process, the 7 priority levels are not enough. The scheduler of the QoS Animator simulates the behaviour of the RMA using only 2 priority levels. We first sort the tasks according to the rate monotonic sort, then we apply the modified utilisation bound test and finally, we simulate the behaviour of the RMA by keeping track of the progress of tasks. These 2 priority levels are used as follows:

- The Scheduler runs at Java' s MAX_PRIORITY.
- The scheduled tasks run at Java' s MIN_PRIORITY
- Runnable tasks are suspended and resumed by the Scheduler

Since JDK1.2 Sun has marked the suspend() method as deprecated, because it is deadlock prone. We safely use it in the QoS Animator by building very carefully the scheduler in order not to refer to any synchronized block. This way, we ensure that the scheduler would be always able to resume any suspended thread, without causing any deadlock.

2.4. Runtime WCET calculation

As the developer of a Java application does not know the target execution environment in advance, the execution time of a task must be calculated to correspond to the runtime executive. In PERC [16] this is being done by analysing the byte-code and charging code structures with amounts of time that corresponds to the runtime configuration. In this work we directly use the run-time executive to measure the execution time of a task, by adapting it to the workload of the system. This solution is specialised for periodic tasks with small periods, such as video and audio play back. In order to calculate the Worst Case Execution Time (WCET) at run-time, we first classify tasks in authentication classes according to their importance, reliability and security, based on a signature they carry. The highest authenticated tasks are thought to be very important (e.g. garbage collector, charging meter etc.), error-free and secure. An authentication class defines the maximum execution time, maximum period and maximum CPU usage that a task is allowed to have and serves the need for a starting point for the WCET calculation. The first effect of this classification is to prevent tasks from overusing the CPU. When a task is first introduced in the system, its WCET is initially calculated according to its period, the maximum execution time and the maximum CPU utilisation allowed by its authentication class. The admission control uses this initially WCET for this task. The task is then scheduled among the other concurrent tasks and its progress is monitored by the QoS Animator. After a policy-defined number of testing iterations, QoS Animator decides on:

- the task' s WCET based on the average execution time achieved during the testing iterations and the maximum execution time allowed by its authentication class
- the conformance of the executed task to its attributes (i.e. its period and its initially given WCET), which depends on the capabilities of the execution environment or any "bad-code" in the task. If such a conformance is not achieved, the task is dropped.

The calculated WCET (equal or lower than the initially assigned) is also fed back to the admission control. Newly arrived tasks might cause existing ones to be dropped, if they belong to higher authentication classes, but this depends on the applied policies.

3. Experimentation and results

In order to evaluate the performance of our prototype we compare it with two other current options. The first uses traditional Java threads and the JVM runs at NORMAL_PRIORITY_CLASS as usual (we call it *Standard Java*). The second is the same with Standard Java but with the JVM running at REALTIME_PRIORITY_CLASS (we call it *RT Standard Java*). Furthermore, in order to evaluate the CPU reservations we achieve, we use the CPU grabber application [14] to simulate concurrent run-

ning time-sharing applications. It runs at NORMAL or HIGH PRIORITY_CLASS as most NT applications.

For our experiments we use a Pentium II 266MHz with Windows NT 4.0 SP3. We also grab the 85% of the CPU with the grabber tool.

3.1. Experiment 1: video playback

In this experiment we measure the inter-arrival times of video frames during a video playback. Each frame is 256x256 pixels and on average 48K. The playback is at 33 fps, i.e. we ideally expect inter-arrival times of 30 milliseconds. Figure 3 represents the unreliable behaviour when using Java as it is for serving applications with time constraints (Standard Java). However, the same result might be produced when using a native process at the 3 lower priority levels of NT. Such a comparison is out of the scope of this work.

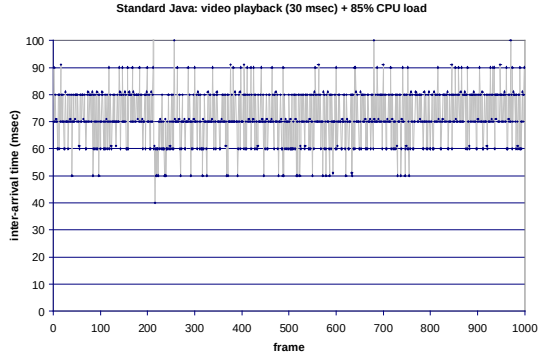

Figure 3

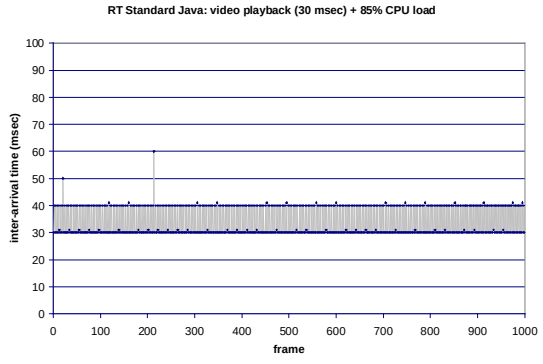

Figure 4

Figure 4 shows the enhancement that can be achieved by boosting the JVM process to REAL-TIME_PRIORITY_CLASS in NT (RT Standard Java). In this case we achieve timeliness equal to the standard time resolution of Windows NT (i.e. 10-15 msec [12]). About 32% of the frames have 10 or more milliseconds delay.

In QoS Animator case (Figure 5), we manage to enhance this timeliness by introducing the Low Frequency Filter. Only 0.5% of the frames have 10 milliseconds delay or more.

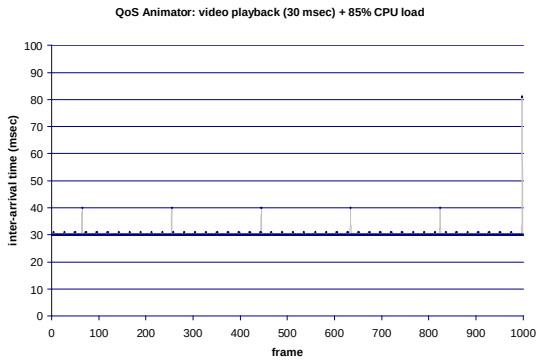

Figure 5

3.2. Experiment 2: multitasking

During the second experiment we measure how the achieved efficiency, showed in experiment 1, can be shared among concurrent tasks. Also, we want to investigate system' s integrity when using the REAL-TIME_PRIORITY_CLASS.

At the first part of this experiment we register 5 tasks with the QoS Animator. Four of them have to do complex logarithmic calculations, with periods of 70, 100, 130 and 200 msec respectively. The fifth task has to calculate 1000 complex logarithms every 100 milliseconds. We assume that the developer has no knowledge of the execution environment (as we target portable real-time code), so she/he defines the tasks according to her/his needs. Furthermore, all tasks demand to repeat their execution for 1000 times. We briefly present here the behaviour of the 100-period task 2 (Figure 6, Figure 7).

The distributions show some statistical differences between those (Figures 8 and 9). The RT Standard Java has a mean value of 111 milliseconds, with 15,2% of the samples out of the range of ±11 milliseconds (Figure 8). The mean value in the QoS Animator case is 102 milliseconds and only 1,8% of the samples have inter-arrival times out of the range of ±11 millisecond (Figure 9). This range depends on the execution time of the 70-period task 1, which is calculated at run-time to be 7 to 10 milliseconds in our system. This illustrates the achieved enhancement in multitasking tasks with time constraints. The shape of the QoS Animator graph corresponds to the RM simulation: up to the first 700 msec (where the 70-period task 1 is running as well), task 2 is always pre-empted by the 70-period task 1. After task 1 finishes, the 100-period task 2 became first in the rate monotonic sorted list and had smoother inter-arrival times.

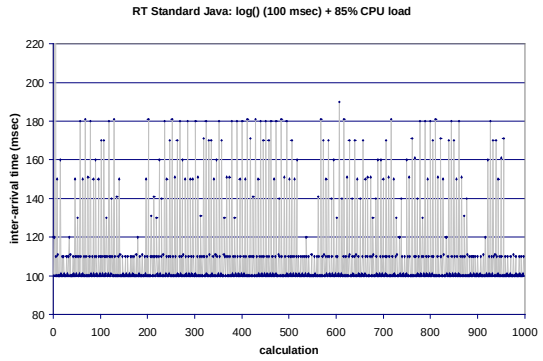

Figure 6

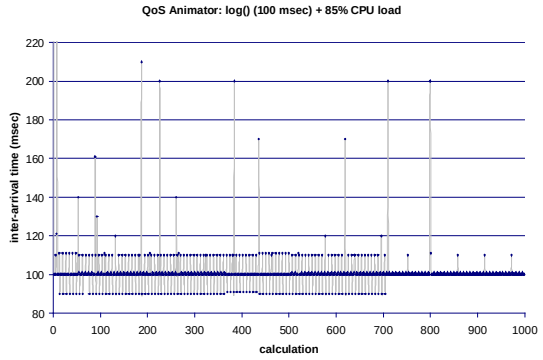

Figure 7

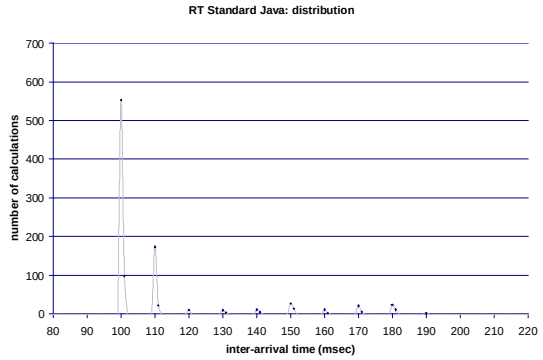

Figure 8

Moreover, another issue that is more important than the produced graphs, is that in the RT Standard Java case the system does not respond regularly. For example, the mouse is moving with about 2-sec pauses between each move. The QoS Animator controls the schedule of the tasks, taking care to leave time to the OS as well. Thus, the system responds normally and moreover, task 5 is detected as a high-CPU-consuming task because it is monitored. The two alternatives that can be followed in QoS Animator to preserve system' s integrity are either to schedule such tasks as best-effort tasks, without caring about their attributes, or to simply drop them. In both cases, the above graphs will be the same. Finally, we replaced the code of task 5 with an infinite loop. In RT Standard Java case the system hanged, while in the QoS Animator case the task is detected and dropped.

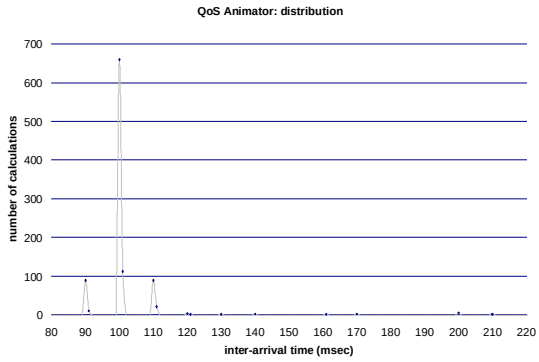

Figure 9

4. Related Work

Lin et al. [17] built a soft real-time server on top of Windows NT, using the REALTIME_PRIORITY-_CLASS of NT. In contrast with our work, this one schedules processes instead of threads (in order to have more range in the priority level), uses the real-time priority of NT "as it is" and requires the execution time of a task to be known prior to execution. Also, no "bad-code" protection is provided and it does not target portable real-time code.

PERC [16][18] is a real-time Java implementation that deals with real-time tasks and their deadlines. As the forthcoming Real-Time Java, it provides a real-time programming language that assumes a skilful developer as it does not deal with cases of "bad-code". Our prototype offers a very simple API and does not require any skills as it handle cases of high-CPU-consumption. In PERC, byte-code analysis is used to calculate the execution time of tasks.

5. Conclusions

We designed a Java CPU manager that provides soft real-time services to Java applications with time constraints. The evaluation of a Windows NT specific prototype proved successful as we managed to enhance the timeliness of NT in the REALTIME_PRIORITY-_CLASS, to enable the portability of real-time code, and to protect the system from untrusted code thus preserve system' s integrity.

6. Acknowledgements

This work was done as part of the dissertation submitted to the University of KENT at Canterbury (UKC) for the degree of M.Sc. in Distributed Systems [19]. It was

also sponsored by BT, as part of the Mware project [20] held in the Distributed Systems group in BT, which explores middleware technologies to enable large scale multicast applications on the Internet.

7. References

[1] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, IETF Network Working Group, Request for Comments 1889, January 1996
http://www.ietf.org/rfc/rfc1889.txt

[2] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, *Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification*, IETF Network Working Group, Request for Comments 2205, September 1997
http://www.ietf.org/rfc/rfc2205.txt

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, *An Architecture for Differentiated Services*, IETF Network Working Group, Request for Comments 2475, December 1998
http://www.ietf.org/rfc/rfc2475.txt

[4] J.Kurose, *Open Issues and Challenges in providing Quality of Aervice Guarantees in High Speed Networks*, ACM Computer Communications Review, vol. 23, no. 1, January 1993, pages 6-15

[5] Paul Menage, *RCANE: A Resource Controlled Framework for Active Network Services*, In Proceedings of First International Working Conference on Active Networks (IWAN'99), Berlin, Germany, June 30 - July 2, 1999, pages 25-36

[6] Philippe Bernadat, Dan Lambright, Franco Travostino, *Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments*, in Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications, Madrid, Spain, December 1998

[7] Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu, *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities*, in Proceedings of the 16th ACM Symposium on Operating Systems Principles. Saint-Malo, France, October 1997, pages 198-211

[8] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda, *Processor Capacity Reserves: An Abstraction for Managing Processor Usage*, in Proceedings of the IEEE Fourth Workshop on Workstation Operating Systems (WWOS-IV), 14-15 Oct. 1993, IEEE Comp. Soc. Press, pages 129-134

[9] Kelvin Nilsen, *Issues in the Design and Implementation of Real-Time Java*, Iowa State University, July 19, 1996

[10] Kelvin Nilsen, *Core Real-Time Extensions for the Java Platform*, J-Consortium, August 19, 1999
http://www.j-consortium.org/rtjwg/draft.8-19.pdf

[11] William Foote, *Real-time Extensions to the Java Platform - A Progress Report*, Fourth International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99), Santa Barbara, California, USA, January 27-29, 1999

[12] *Real-Time Systems and Microsoft Windows NT*, Microsoft Developer Network Library, Microsoft Corporation, June 29, 1995
http://msdn.microsoft.com/library/backgrnd/html/msdn_realtime.htm

[13] C.L.Liu and J.W.Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*, Journal of the ACM, vol. 20, no. 1, 1973, pages 46-61

[14] Alan Smith and Richard Jacobs, *Quality of Service Management within a Middleware for Large Scale Multicast Applications*, IEEE Workshop on QoS Support for Real-Time Internet Applications, Vancouver, British Columbia, Canada, June 2-4, 1999, pages 85-92

[15] Lei Zhou, Kang G. Shin and Elke A. Rundensteiner, *Rate-monotonic scheduling in the presence of timing unpredictability*, in Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium, 3-5 June 1998, IEEE Comput. Soc, pages 22-27

[16] Kelvin Nilsen and Steve Lee, *PERC Real-Time API (Draft 1.3)*, NewMonics, 1998
http://www.newmonics.com/pdf/perc_api.pdf

[17] Chih-han Lin, Hao-hua Chu, Klara Nahrstedt, *A Soft Real-time Scheduling Server on the Windows NT*, 2nd USENIX Windows NT Symposium, Seattle, WA, August 1998, pages 157-166

[18] K.Nilsen, S.Mitra, S.Sankaranarayanan, and V.Thanuvan, *Asynchronous Java exception handling in a real-time context*, in Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications, Madrid, Spain, December 1998

[19] Gerasimos Xydas, *Real-Time Java for End-System QoS Control*, a dissertation submitted for the degree of M.Sc. in Distributed Systems to the University of KENT at Canterbury, 17 September 1999

[20] Mware project, BT,
http://www.labs.bt.com/projects/mware